National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

The University of Alberta

A Natural Language Interface to a Robotic Work Cell

by

Simon P. Monckton

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of

Master of Science

Department of Mechanical Engineering

Edmonton, Alberta

Spring, 1989

National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Service des thèses canadiennes

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-52794-3

Canada

The University of Alberta

Release Form

Name of Author: **Simon P. Monckton**

Title of Thesis: **A Natural Language Interface to a Robotic Work Cell**

Degree: Master of Science

Year this degree granted: Spring 1989

Permission is hereby granted to **The University of Alberta Library** to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly, or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive tracts from it may be printed or otherwise reproduced without the author's written consent.

....

(Student's signature)
Simon P. Monckton
9214-117 St.
Edmonton, Alberta
T6G 1S2

Date : April 26, 1989

The University of Alberta
Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled

A Natural Language Interface to a Robotic Work Cell
submitted by
Simon P. Monckton

in partial fulfilment of the requirements for the degree of

Master of Science.

Roger W Toogood
(Supervisor)
R. Goebel
J. Sprague
D. Steigmann

Date : 24 April 89

To my wife, Simone, for her patience and encouragement and to my Parents, Jean and George, for their faith and support.

Abstract

This dissertation describes the design and implementation of a prototype natural language interface to a small work cell equipped with a simple binary vision system. Installed on a personal computer, this TSKMSTR software package investigates the minimum requirements of natural language processing by relying on syntactic and semantic conventions to develop a command template. These templates, once incorporated into a Horn Clause task description, are executed by a meta-PROLOG interpreter. Further, this PROLOG representation permits the implementation of task generalization during an English teach dialogue, a facility currently unavailable on other natural language work cell interfaces. In addition an homogeneous transformation world model has been designed that places traditionally explicit world model management duties into a hidden, run-time module. An evaluation of the TSKMSTR system and recommendations to improve the current software are also provided.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Dr. Roger Toogood, for his friendship, support and advice during the course of this project. Further I wish to express my gratitude to the the University of Alberta Department of Mechanical Engineering for their financial support throughout my studies.

Contents

Abstract v

Acknowledgements vi

1 Introduction 1

1.1 The Human Perspective 1

1.2 Robot Emancipation 3

2 World Modelling, Task Representation and Robot Programming 5

2.1 A World Model Hierarchy of Robot Control Languages 5

2.1.1 Leadthrough Programming : An Explicit World Model 6

2.1.2 Robot Level Programming : an Explicit World Model 8

2.1.3 Object Level Programming : an Implicit World Model 16

2.1.4 Task Level Programming : an Implicit Task Description . . . 22

2.2 Industrial Off-Line Programming Languages 23

2.2.1 Commercial Languages 23

2.2.2 Research Robot Control Languages 36

2.2.3 Comparison of Modern Robot Programming Strategies 40

3 Natural Language Processing and Robot Programming 43

3.1 Natural Language Processing Methods 43

3.1.1 What is Natural Language? 43

3.1.2 Symbols 44

3.1.3 Syntax 45

3.1.4 Semantics 51

3.2 Automating Natural Language Understanding 52

3.2.1 Divide and Conquer Approaches 52

3.2.2 Unified Approach 57

3.3 Robot Programming with Natural Language 62

3.3.1 SHRDLU : Winograd, 1972 62

3.3.2 A High-Level Hierarchical Robot Command Language (HIROB) : Bock, 1984 63

3.3.3 A Natural Language Interface for an Instructable Robot : Maas and Suppes,1985 65

3.3.4 A Natural Language Interface to a Robot Assembly System : Selfridge et al., 1986 67

3.4 The Bare Essentials of Natural Language Interfaces for Robots 69

4 The Experimental Work Cell 72

4.1 The Manipulator 72

4.2 The Vision System 74

4.2.1 The System's Physical Attributes 74
4.2.2 Vision System Operation 74
4.2.3 The Vision Processor: VISYS 80

5 The Natural Language Interface **91**

5.1 TSKMSTR Operation 91
5.1.1 Example Dialogues 94
5.2 TSKMSTR Design and Implementation Overview 105
5.3 The Scanner and Parser 106
5.3.1 The Grammar 106
5.3.2 Vectors 107
5.3.3 Prepositions 108
5.3.4 The Parser Generator 113
5.4 The Semantic Processor 119
5.4.1 The Semantic Processor Registers 120
5.4.2 Register Collection and Distribution 121
5.4.3 An Example 129
5.5 The Command Interpreter 131
5.5.1 Teach Mode 135
5.5.2 Unsupported Legal Syntactic Constructs 137
5.6 World Model, Kinematic Engine and Task Primitives 138
5.6.1 World Model and Kinematic Engine 138

5.6.2 Task Primitives 147

5.7 Communication and Support Utilities 151

6 TSKMSTR Evaluation, Recommendations and Conclusions 152

6.1 Natural Language Interface Performance and Evaluation 152

6.1.1 Scanner and Parser Evaluation 153

6.1.2 Semantic Processor Evaluation 154

6.1.3 The Command Interpreter 157

6.1.4 World Model Performance 160

6.2 Hardware Evaluation 162

6.2.1 Manipulator Performance 162

6.2.2 Vision System Performance 163

6.3 Recommendations for Future Research 164

6.3.1 Modifications to TSKMSTR 164

6.3.2 Other Approaches to Natural Language Interfaces and Work Cells 166

6.4 A Final Comment on Programming Robots in English 166

6.5 Conclusion 167

References 170

A Robot Kinematics and Modelling 176

A.1 Position 177

A.2 Orientation 178

A.3 The Homogeneous Transform 180

A.4 Homogeneous Transforms and Robot Kinematics 181

B Binary Vision Processing Algorithms **186**

B.1 The Moments of Area 186

B.2 Moment Invariants 188

B.2.1 Scale 188

B.2.2 Rotation 188

B.3 VISYS Shape Descriptors 188

B.3.1 The Area and First Moment Invariant, I_1. 189

B.3.2 Density 191

B.3.3 The Perimeter Length, Number of Holes and Compactness . . 191

C An Explained Dialogue Session **195**

D The BNF Grammar for the TSKMSTR Natural Language Interface. **200**

List of Tables

2.1 Currently Available Robot Control Languages 24

2.2 The IRDATA standard data types condensed from Blume and Jakob [14] *Programming Languages for Industrial Robots.* 27

2.3 A comparison of current robot control language flow control. 28

3.1 A comparison between CD slots and Semantic Roles, based on Allen [56] and Schank and Riesbeck [57] 58

3.2 An Example of Conceptual Analysis on: *Fred gave Sally a book.* condensed from Birnbaum and Selfridge [57]. 61

5.1 Bennett's [52] analysis of TSKMSTR's locative prepositions. 111

5.2 Bennett's [52] analysis of TSKMSTR's non–locative prepositions. . . 111

5.3 A comparison between TSKMSTR and traditional English word classes 116

5.4 The Type Tags employed by TSKMSTR's semantic processor. 123

5.5 A list of the TSKMSTR primitive work cell functions. 148

A.1 Alternate Representations of Position and Orientation 185

[illegible] Backus Naur Form for the Natural Language Interface. 201

[illegible] Backus Naur Form for the Natural Language Interface (cont.) 202

D.3 Backus Naur Form for the Natural Language Interface (cont.) 203

D.4 Backus Naur Form for the Natural Language Interface (concluded). . 204

List of Figures

2.1 A Typical Work Cell environment 10

2.2 An example of a "tree" like object model. Each arc is an homogeneous transformation and each node is a coordinate frame. 17

2.3 A portion of an Assembly Graph produced by using the "tree" structure for object modelling, the dashed lines indicate connections between parts and the arrows indicate the direction of the linking transforms. 18

3.1 A Simple Sentence Grammar in BNF notation 46

3.2 An Transition Network for a Simple Sentence 47

3.3 A Regular grammar generating alphabetical combinations of *a*, *b*, *c*, *d* and *e* 49

3.4 An STN generating alphabetical combinations of *a*, *b*, *c*, *d* and *e* . . . 49

3.5 A CFG producing equal numbers of *a* and *b* 50

3.6 An RTN producing equal numbers of *a* and *b* 50

3.7 A Context Free Grammar for a Transitive English Sentence 54

3.8 A Semantic Grammar fragment from LADDER [19] 59

4.1 A schematic of the University of Alberta's prototype work cell. 73

4.2 The VISYS binary vision system flow diagram. 75

4.3 The Local mode image acquisition phase. 77

4.4 The Local mode image analysis or 'features' phase. 77

4.5 A typical Shape Edit session screen. 78

4.6 A typical ORIENT orientation method analysis phase. 78

4.7 The Hole, Radii and Principal Axis orientation methods used in the VISYS vision system. 89

4.8 The evolution of a typical perimeter in a VISYS image. 90

4.9 The Modified Circle Method or Symmetry Method as employed by VISYS. 90

5.1 A flow diagram detailing the global operation of the TSKMSTR system. 95

5.2 Typical user command. Note that the vision system has been invoked. 97

5.3 Vision data, returned in the Messages window, is used in the execution of the user's command 97

5.4 With the gripper affixed to the disk, the disk may be moved relative to it's current position. 98

5.5 New labels may be made in the work cell through the `assign` task primitive. Objects may then be attached to these sites. 98

5.6 An example of the `roll` primitive with the adverb `anticlockwise`. The processor also accepts signed degree angles. 100

5.7 An example of a compound sentence. Note the execution order implied by the task equations reported in the Interface window. 100

5.8 A more complex example of a compound command clause. 100

5.9 A worst case example of compound clauses. Once again, note the execution order. 101

5.10 An example of the `find` primitive. 101

5.11 The Teach mode is invoked. 103

5.12 The query engine asks the user to generalize the command. 103

5.13 During the definition of `insert` the Teach mode is reinvoked by a `grasp` command, not defined in this example. 104

5.14 `Grasp` is defined. Note the inclusion of the `affix` command relating the gripper to the head of bolt1. 104

5.15 The teach mode 'ascends' back into the `insert` teach session. 104

5.16 The parse structure for *Completely affix the peg to the hole*. 118

5.17 The structure of *command template* used in the Semantic Processor . 127

5.18 KIE's affixment notation for a stack of blocks. 144

5.19 A complete Assembly Graph as modelled in KIE for the objects shown in figure 2.2. 146

B.1 A hole perimeter in camera coordinates. Arrows indicate direction of perimeter list. 193

Chapter 1

Introduction

When robots were first introduced to industry by the fledgling Unimation in 1961, they were taught new tasks by hand. Thirty years later, with few exceptions, the industrial robot relies on much of its understanding of the world through a hand guided teaching cycle. Though sensors such as vision and touch are available to add flexible behaviour to the robot's innate mechanical versatility, industry has continued to rely on the less flexible manually taught machines. One of the major reasons for industries' caution stems from the difficulty and expense of routinely programming a robot within a *work cell*, or robot assembly station, equipped with a complex array of sensors. The experienced assembly technician must undergo considerable training or describe the assembly task to a full-time programmer who may have little understanding of the assembly task.

To discover the fundamental problem of bridging the gap between the languages of the robot and the assembly technician first requires a discussion of how engineering assumptions have traditionally molded machine behaviour and how the modern robot's behaviour is molded, not by the engineer, but by the technician on the shop floor.

1.1 The Human Perspective

Like an artist's perspective is instilled in his painting, the engineer's impressions of the world are preserved in the design and operation of his inventions. The designer's assumptions concerning how, where, when, and for what purpose his invention is to be used, will form its final configuration and operation in the work place. From the stone axe to the drill press, the designer must make assumptions about the intended placement and use of a prospective machine. In this sense the machine becomes an expression of the designer's view of the working environment. Discrepancies between this assumed environment and the real world provide a means of measuring the machine's flexibility. The dependence on these assumptions and the tolerance of deviations from them, while still producing the intended results, define that flexibility. Should the machine demand a tightly regulated environment, this, too, is often incorporated into the design. For example, some of the first woodworking lathes were manually operated and the cutting tools hand held. Modern lathes, working harder materials, use 'built- in' precision cutting tools operated through calibrated mechanical screws, an example of regulating the lathe's world.

In the industrial setting, efficient, high volume production often requires dedicated machines working within tightly controlled environments. For instance, assembly lines often use conveyors to transport material from one operation to another, while elaborate shakers and feeders position and orient material for processing. Despite this rigid structuring, production lines invariably possess disordered environments demanding mechanisms capable of flexible behaviour. In this situation, human operators often guide specialized tools to complete the task, thus closing the gap between dedicated machinery and less structured assembly operations. Ultimately a task may either become so complex, the environment so unpredictable, or the quantity so small that only a skilled craftsman is sufficiently equipped with the flexibility of judgement and precision of operation to adequately and efficiently complete the assembly.

1.2 Robot Emancipation

The robot, like any other machine, has been designed by engineers with a unique set of assumptions that define the machine's general behaviour. These assumptions, however, do not limit the application of the robot. In fact, the Robotic Industries Association defines the robot as follows:

> An industrial robot is a reprogrammable, multifunctional manipulator designed to move materials, parts, tools, or special devices through variable programmed motions for the performance of a variety of tasks.

Unlike previous industrial machines the robot is totally programmable, having no single predetermined operational definition. The programmable robot's operational characteristics are designed and altered by shop floor personnel. The manipulator's precise behaviour becomes that of the 'teaching' technician who, in effect, impresses his work methods and environment onto the robot.

Despite this apparent versatility, the majority of robots developed within the last thirty years remain employed in highly structured assembly line environments. Most industrial robots perform essentially "pick and place" tasks, while few execute "intelligent" decision-based assembly operations. One reason for this, of course, is that most robots are employed in human working environments dominated by human assembly methods. Another fundamental reason is that although robots are programmable, the majority of hand taught industrial robots are no more flexible than any other dedicated machine once the programming is complete. The robot's world is shackled to the operator's impressions of the work place, regardless of the consequences.

To free the robot from the limits of a rigid and potentially inaccurate human world model, sensors and off-line programming languages have been incorporated into the modern robot's array of capabilities. These allow the robot to base its behaviour on observed changes in the work environment, in effect forming its *own* impression of the world. Despite these added advantages, the introduction of these flexible robots has been difficult. The complexity and expense of routinely 'translating' the programmer's intentions and world impressions to robot level equivalents is often difficult to justify.

In summary, machine behaviour can be interpreted as an image of a designer's world model. However, to take full advantage of the robot's mechanical flexibility, its behaviour should be based on an internal world model. Unfortunately, the difficulties of incorporating the assembly technician's intentions into the flexible robot's world model are considerable. Until such time as the robot is capable of forming its own assembly plans, some means of efficiently and economically translating these intentions must be found. Though many notable attempts have been made to simplify the 'communications gap' between man and machine such as AUTOPASS [8] and AL [9] programming languages, the majority of modern robot languages are either primitive off-line versions of the on-line hand teaching protocol or sophisticated PASCAL or FORTRAN-like programming languages requiring full-time programming expertise.

This thesis will discuss, in detail, the difficulties surrounding modern robot programming and present one potential solution to the problem of communicating with robots: a PC-based English natural language interface to a robotic work cell.

Chapter 2

World Modelling, Task Representation and Robot Programming

The concept of world modelling is often considered academic and theoretical in most discussions on robotics and is only lightly treated at the undergraduate level [32]. This is an unrealistic judgement, however. Throughout the history of machines, engineers have spent their educational years developing their own world models describing material and physical behaviour, subsequently applying them to the tools around us. The topic of world modelling in manufacturing is of growing importance as finite element analysis, computer aided design and computer aided manufacturing developers strive to integrate modelling methods, thus speeding the design and production process. The development of flexible or intelligent work cells demands the incorporation of this modelling ability in robots. The following discussion reviews the methods employed for work cell modelling and its impact on task representation and robot programming.

2.1 A World Model Hierarchy of Robot Control Languages

In its most rudimentary form, robot programming consists of prescribing a sequence of robot manoeuvres and tool operations, perhaps in conjunction with external events. These manoeuvres and events are monitored and coordinated by some controller. A controllers capabilities, like those of the manipulator, are closely linked with the representational scheme adopted to model the actions and events within the environment. As the demands made on a work cell rise, a capable representational scheme or language must be found to represent the environment.

This section will look at world modelling techniques, task representation and robot programming languages. Four broad categories characterize the programming philosophies commonly found within modern work cells:

1. Leadthrough programming : an Explicit World Model
2. Robot Level programming : an Explicit World Model
3. Object Level programming: an Implicit World Model
4. Task Level programming : an Implicit Task Definition

These four methods are not necessarily distinct, rather they exemplify a steady increase in flexibility and complexity of work cell programming methods.

2.1.1 Leadthrough Programming : An Explicit World Model

The original and dominant form of robot programming, known as leadthrough programming, amounts to teaching the robot "by showing". Most robots are taught by "hand guiding", storing an operators knowledge for each task description during an on-line teaching session. Consequently, robot operations are defined by lists of memorized joint positions. Though still reliant on a teacher's set of external assumptions, in effect the machines's description of the world has been raised from tangible mechanical stops and switches to the less tangible *joint space.*

Leadthrough based systems tend to be relatively inexpensive and simple to operate. With a small keyboard *teach pendant* or by directly grasping the end effector, an operator guides the end effector to a desired point, fine tunes the effector's position and orientation, and saves the point. Similar teaching methods include scale *master* arms guiding the *slave* work cell manipulator through required motions or moving the end effector through a task by hand. Unfortunately these methods are time consuming and often difficult to debug, the result being that the systems are not reprogrammed often, reducing their potential flexibility.

Depending on the complexity of the system, the saved points or *knot points* may be saved in joint space or in *cartesian space.* This more sophisticated form of leadthrough system, often equipped with path generation software, stores the target position as a coordinate frame, computing an optimized path prior to run-time.

The paths developed in early leadthrough systems were products of *point-to-point* control. Point-to-point trajectories between knot points are generated when the joints are allowed to move as rapidly as possible to their final position. Though they make little computational demand on the controller, the path of the tool tip is often unpredictable. Path control is necessary for assembly and finishing where small tolerances and high quality are priorities. More precise methods such as *joint interpolated* control or *cartesian interpolated* control require the introduction of a microprocessor to the work cell environment.

Joint interpolated motion between initial and final joint positions are motion instructions issued such that all joints arrive at the final position simultaneously. Joint interpolation is computationally inexpensive and rapidly produces a predictable, though complex, tool path in the work space.

Cartesian interpolation, designed to produce a predictable linear path through cartesian space, places a heavy computational load on the controller and often requires large joint accelerations or torques. This method is usually employed only when high precision is required in tool positioning, usually during the final approach and actual machining or assembly operations.

Obviously, robot manoeuvres rarely occur in isolation and must be coordinated with external events such as material arrival or process completion. Signals convey this information to the controller and trigger a predetermined action sequence. These signals are incorporated into the program through the controller. Some hand guided robots can interact with their environment through simple sensor input [7], such as touch or force sensing.

The dominant advantage of leadthrough programming is simplicity. Since this method can employ relatively unskilled programmers, the hand taught robot is popular in industry. If the robot's role is well chosen, there is often little advantage in employing more sophisticated robots.

A significant disadvantage of this method is that any new procedure must be virtually retaught from the beginning. The concept of subroutines is both limited and difficult to employ, often more difficult than simply retraining the robot. Furthermore, since the operator develops the robot environment on the shop floor by hand, the assembly line is interrupted and production stopped. Deficiencies of this method appear when operations employing precise path control and event based decisions are given to hand taught work cells. Though some are equipped with simple sensor-based decision making functions, the controller's lack of computational power limits their decision making capacity. Often the only solution is the introduction of a small microcomputer to control the work cell. An off-line programming language then orchestrates work cell processes.

2.1.2 Robot Level Programming : an Explicit World Model

The simplest form of off-line programming retains the leadthrough method of joint space task representation. These positions are stored in data files and called during program execution by a MOVE type instruction. While these explicit robot configurations precisely describe manipulator motions, simple flow control such as GOTO or IF ... THEN commands provide a limited degree of flexibility. Variables are limited to simple primitives such as integers and reals. Motion commands tend to be limited to point-to-point movement from one configuration to another. Path control amounts to multiple move commands through intermediate robot configurations. The influence of sensor information on task execution remains essentially boolean switch information.

A variation on this method employs a cartesian space representation, by introducing homogeneous transformations into the programming environment. See appendix A or Paul [22] for more information on homogeneous transforms.

World Modelling with Homogeneous Transformations

In Paul's discussion on modelling with homogeneous transformations, T_n is the transformation from the base of the manipulator to the gripper and represents the forward kinematic solution. The position of the end effector, a gripper or specialized tool (e.g. a drill), may be described by the homogeneous transformation TOOL relative to the final joint. The displacement of the manipulator base from the origin of the work cell may be described by an additional transformation E. The expression defining the position, G, of the end effector relative to the work cell then becomes:

$$\mathbf{G} = \mathbf{E}\ \mathbf{T}_n\ \mathbf{TOOL}$$

Like the manipulator, assemblies can be interpreted as kinematic chains within the work cell. For example in figure 2.1.

- the transform FEEDER relates the position of a feeder mechanism in work cell coordinates.

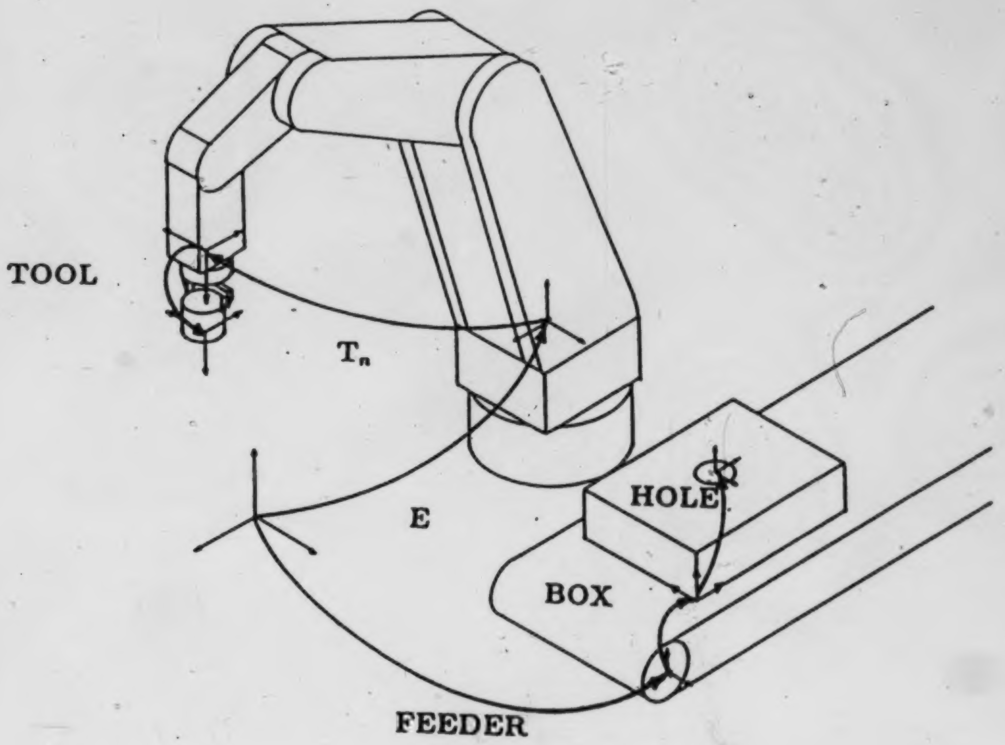

TOOL
T_n
E
HOLE
BOX
FEEDER

Figure 2.1: A Typical Work Cell environment

- BOX defines the location of the box in FEEDER coordinates.
- HOLE defines the position of a hole in BOX coordinates.

Therefore:

$$\mathrm{L} = \mathrm{FEEDER\ \ BOX\ \ HOLE}$$

defines the location of a hole in work cell coordinates.

These transformations specifying relative object locations in a work cell can be used to 'calibrate' the work cell during the programming phase. Though they may be derived from design drawings, usually the robot manipulator and forward solution are used to compute the values of unknown transformations. For example, if the FEEDER transform is not known, the following expressions may be used to determine its value:

$$\mathrm{E}\ T_n\ \mathrm{TOOL} = \mathrm{FEEDER\ \ BOX\ \ HOLE}$$

Solving for FEEDER...

$$\mathrm{FEEDER} = \mathrm{E}\ T_n\ \mathrm{TOOL\ \ HOLE}^{-1}\ \mathrm{BOX}^{-1}$$

where T_n is the forward solution using manipulator joint angles gathered when the tool is placed on the site of the hole.

Unfortunately, the use of the manipulator as a tool to determine object positions during programming interferes with the assembly line operation and is little better than hand teaching the robot. An alternative is the use of sensors during work cell operation to determine the missing transforms. A vision system with known camera location CAM, relative to the work cell, determines the BOX position, BOXPOS, in the work cell:

$$\mathrm{BOXPOS} = \mathrm{CAM\ \ BOX}$$

Since CAM is known, the transform BOX is expressed as:

$$\text{BOX} = \text{CAM}^{-1}\ \text{BOXPOS}$$

It should be noted that this approach removes the responsibility of world model management from the programmer and places it firmly into the run time environment of the work cell.

World Modelling and Task Representation

If the above method of manipulator and assembly modelling is employed to represent assembly operations, a task can be described in work cell coordinates by a sequence of homogeneous transform equations. Solving these equations for T_n expresses a task in terms of manipulator position and orientation. Applying the Inverse Kinematic solution to T_n produces the requisite robot joint angles.

Using the earlier example, the drill placement at the sight of a future hole in the box is equivalent to:

$$\text{E}\ T_n\ \text{TOOL} = \text{FEEDER}\ \text{BOX}\ \text{HOLE}.$$

Solving this equation for T_n

$$T_n = \text{E}^{-1}\ \text{FEEDER}\ \text{BOX}\ \text{HOLE}\ \text{TOOL}^{-1},$$

isolates the manipulator and expresses the position of the end effector in terms of the relative locations of the tool, work cell, and assembly. The Inverse kinematic solution applied to T_n will produce those joint angles needed to bring the drill to the site of the hole.

Though the subsequent world model and order of execution continues to be explicitly expressed in the program, three significant advantages stem from the adoption of this new description:

1. **Legibility**

   The immediate advantage of this method is that end effector positions, previously occurring as joint angle lists, appear in familiar cartesian coordinates with some orientation quantities such as Roll, Pitch, Yaw (RPY) or Euler parameters. This greatly improves the coding and debugging processes that arise within the design phase of a robot program.

2. **Path Generation and Control**

   Paths through three dimensional space can be generated and executed from within the program. Intermediate points along the chosen path can be described as mathematical functions, useful for optimizing robot efficiency or load carrying performance. This scheme enables collision avoidance and target tracking algorithms, since paths can be developed to compensate for environmental conditions.

3. **Generalization of Common Subtasks**

   Since this method allows variable end effector and object positions, common or repetitive subtasks can be represented as subroutines with variable parameters. These parameters can be provided by sensor systems such as vision or touch.

When the cartesian world model is combined with homogeneous transforms, as outlined earlier, robot motion depends on the solution of a Task equation for T_n. These explicitly stated equations, solved at run-time, allow for the resolution of positions within the program through sensors.

Work cells equipped with vision systems may explicitly call upon visual information to provide unresolved position data, while other systems rely on a teach phase prior to final program installation. In this phase undefined coordinate frames are described with the aid of a teach pendant or master arm. Once the teach session is complete the program is compiled and ready to run [11].

Task Representation and Programming

A variety of data structures are used to model these equations. To justify the use of certain data structures requires an examination of the assembly environment. An assembly composed of many parts can be viewed as either:

- a collection of dissociated parts uniquely related to the work cell origin, or
- a coherent association of parts uniquely related to one another.

In addition to these considerations the movement of an assembly must be reflected in the movement of the sum of its parts.

Small assemblies may be treated as a group of dissociated features and parts, though this requires cautious bookkeeping by the programmer. Consequently small assembly programs often use 4 × 4 arrays to represent the homogeneous transform and explicitly alter part locations when necessary. Large assemblies, however, render this approach an inefficient and error prone programming technique.

A partial answer to this problem is to minimize the bookkeeping required during the program development by associating parts and features with one another. This is done by defining within each part a central coordinate system; the useful features that appear on that part are then explicitly described relative to that central coordinate frame.

The result is that a part appears as a tree–like structure with the central frame acting as the root and the part features as branches. The location of a part feature then becomes the product of the part's central frame location and the feature's relative location. An assembly of two parts is the *attachment* of one or more branches of two trees. The location of any part on an assembly can be found by determining an appropriate kinematic chain from the work cell origin to the part through that assembly and finding the product of the chain's components.

Another representation useful for this form of world model, is the use of records. In this way a part and its features can be represented together in a single data structure. A typical data structure for a can of soup might be:

```
type
vector = array [1..3] of real;

feature = record of
                  Name        : string;
                  forward_XYZ,
                  forward_RPY: vector;
                  inverse_XYZ,
                  inverse_RPY: vector;
             end;

tin_can = record of
                  central,
                  top,
                  bottom,
                  grasp_posn,
                  approach_posn : feature;
           end;

var
soup : tin_can;
```

Where `central` is the central frame of the tin can. The `forward_` and `inverse_` prefixes denote transforms from `central` to the feature and from the feature to central respectively. Since the rotation matrix is not an intuitive description of orientation, rotation often appears in its RPY or Euler form. The advantage of this approach is that only a small set of `central` frame locations need be maintained explicitly by the programmer. See figure 2.2 for an example of this technique.

Even with this form of nested data structure, the problem of bookkeeping still remains — the task equation must be assembled by hand and the world model explicitly maintained within the program. As larger assemblies increasingly complicate the world model, the programmer may consider automating the maintenance and construction of task equations, a characteristic feature of Object Level languages.

2.1.3 Object Level Programming : an Implicit World Model

When the tree–like representation of individual parts, described in the previous section, is employed to describe an assembly of many parts, a graph structure results similar to figure 2.3. A graph structure is similar to a tree in that each node has multiple daughter nodes. Unlike the tree structure, however, the graph also has multiple parent nodes. This means the graph often possesses multiple routes from one part or node to another. In this model the entire work cell becomes a single graph, rooted in the world coordinate system. Any component in the model, such as a robot gripper or part feature, ultimately stems from the world system. In the homogeneous transform modelling method, described in section 2.1.2, the motion of a robot to a position in space is equivalent to the closure of two kinematic chains. This closure is equivalent to the graphical conjunction of two branches of the world graph i.e. the tooled robot and the target assembly.

The construction of a task equation will require some search strategy originating from the world node to establish the two appropriate kinematic chains that, when equated, will describe the conjunction of these two coordinate frames. Assemblies are

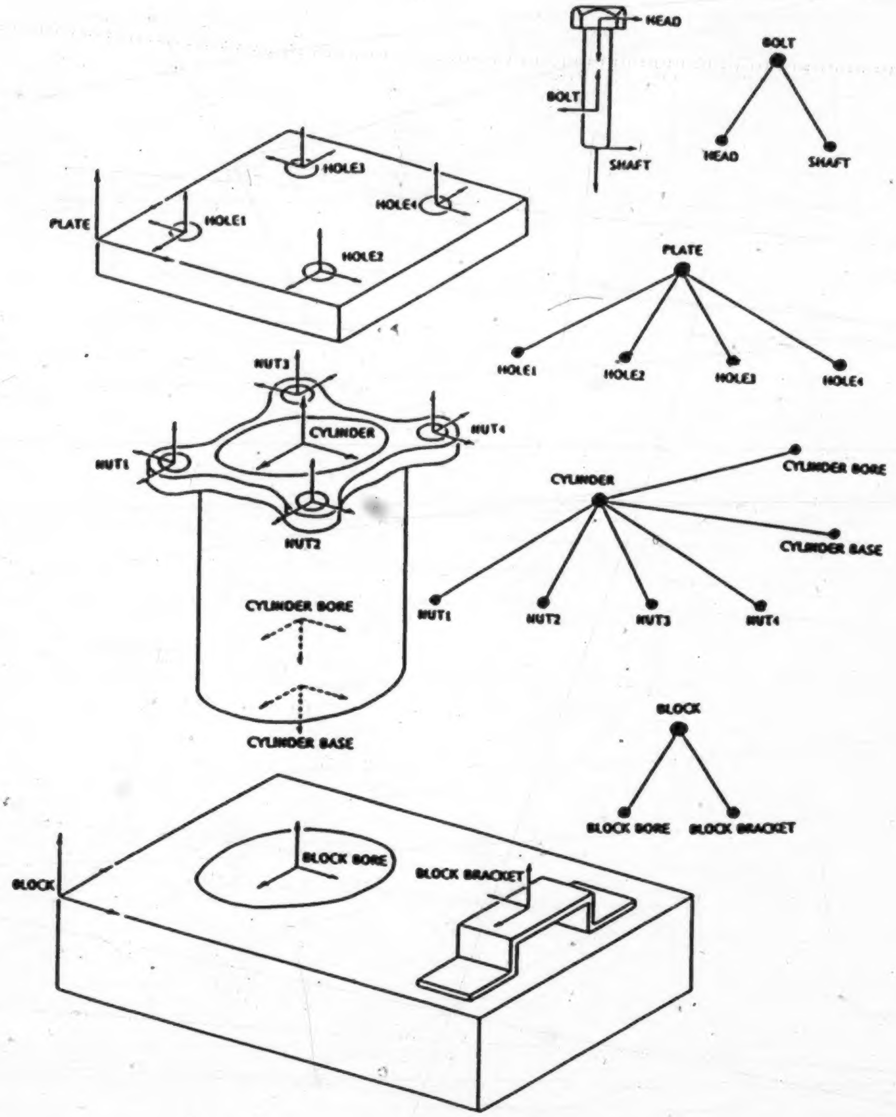

HEAD
BOLT
SHAFT
BOLT
HEAD
SHAFT
HOLE3
HOLE4
PLATE
HOLE1
HOLE2
PLATE
HOLE1
HOLE2
HOLE3
HOLE4
NUT3
CYLINDER
NUT4
NUT1
NUT2
CYLINDER BORE
CYLINDER BASE
CYLINDER
CYLINDER BORE
CYLINDER BASE
NUT1
NUT2
NUT3
NUT4
BLOCK
BLOCK BORE
BLOCK BRACKET
BLOCK
BLOCK BORE
BLOCK BRACKET

Figure 2.2: An example of a "tree" like object model. Each arc is an homogeneous transformation and each node is a coordinate frame.

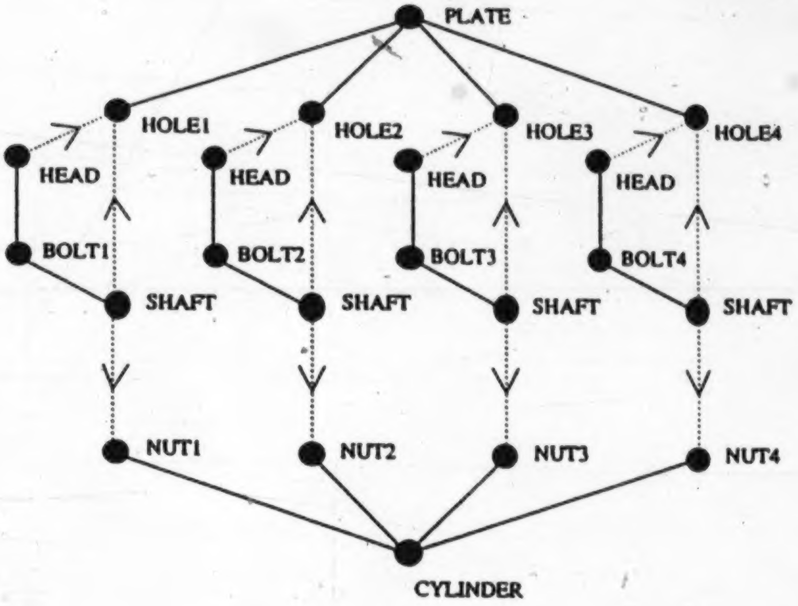


PLATE
HOLE1
HOLE2
HOLE3
HOLE4
HEAD
HEAD
HEAD
HEAD
BOLT1
BOLT2
BOLT3
BOLT4
SHAFT
SHAFT
SHAFT
SHAFT
NUT1
NUT2
NUT3
NUT4
CYLINDER

Figure 2.3: A portion of an Assembly Graph produced by using the "tree" structure for object modelling, the dashed lines indicate connections between parts and the arrows indicate the direction of the linking transforms.

built through the cumulative attachment of subassemblies, also modelled as kinematic chains.

Data structures for this model facilitating a search strategy typically result in multidimensional linked lists. A typical structure in pseudo-PASCAL might be

```
type
        feature = record of
                            Part_Name,
                            Feat_Name : string;
                            XYZ,
                            RPY : vector;
                            Next_Feature,
                            Attached_To : pointer to feature;
                          end;
```

The `central` frame is no longer mentioned explicitly, acknowledging the importance of the part features in assembly and relegating the `central` frame concept to an implicit reference point in the part. This two dimensional structure allows access to both the feature list of a part (by reviewing feature structures through the `Next_Feature` pointer) and the list of attached parts (by reviewing subsequent structures through the pointer `Attached_To`). The bookkeeping operations required for less sophisticated models have been replaced by assembly search functions. The location and position of any part or feature is the product of the appropriate kinematic chain found using a search algorithm.

Utility functions for joining and severing coordinate frames are required, enabling parts to be attached to, and detached from one another. Care must be taken to reflect the real world in the description of these attachments. Depending on the form of real world attachment a part may or may not be dependent on its neighbouring part for positional information. For example :

> In the real world, if block A is on top of block B and block B is moved, then block A will move also. However, If block A is moved, B will remain stationary unless blocks A and B are mutually attached.

This form of modelling requires that either the work cell is fully aware of real world conditions such as gravity, coefficients of friction, and object dynamics or that the programmer explicitly state the form of object attachment. At this level of modelling an explicit attachment function is usually employed to modify object interdependence.

Robot motion is described through a series of conjunctions between tool and target coordinate frames or *objects*. The construction of the task equations governing the conjunction is virtually hidden from the programmer. Since the world model resides implicitly in the program, neither the manipulator nor parent assembly need be explicated while matching tool and target features. Further, sensor systems previously called by the programmer may be incorporated and hidden inside the world model [29]. Sensor access can be initiated and orchestrated by the world model when insufficient world model data inhibits the completion of a robotic task. For example a vision system might be enabled when an object is specified that does not exist in the world model. Alternatively the sensors may function continuously, updating the model in real time.

A classic example of an implicit world modelling system is the AL world model [10]. This model is virtually completely hidden from the user save for the AFFIX and UNFIX commands. The AL model employs these affixment functions to declare spatial interdependencies between coordinate frames. The affixment structure developed by AL becomes a graph when multiple assemblies are affixed to one another. Each node in the graph is composed of a frame with a pointer to a *calculator list*, a set of *calculator nodes* (not to be confused with graph nodes).

Each calculator node describes an affixment relationship between two frames. An affixment arc between two graph nodes (or frames) is modelled by *two* calculator nodes one from each frame's calculator list. Each calculator node is composed of three boolean fields and a frame pointer described as follows:

Boolean Affixment Field : this is used to indicate the spatial dependency (Rigid or Nonrigid) between the two linked frames.

Boolean Parent Field : this indicates the directional sense of the transformation indicated by the *Boolean Validity Field* described below.

Boolean Validity Field : briefly, this indicates whether or not the existing location for the frame is valid. Otherwise the location of the frame is computed (with continuous reference to the directional sense implied by the Parent field).

Pointer Transformation Field : points to the affixed frame.

The bidirectional system of calculator nodes for each affixment gives the model the ability to determine and classify the spatial interdependency between coordinate frames as *Rigid* (block B moving block A) and *Nonrigid* (block A moving free of block B).

The representation and maintenance of object locations within the world model is virtually hidden from the programmer, but implied in the user's program. Ultimately the *task definition* becomes an explicit series of actions or operations on object features (represented by coordinate frames) instead of precise robot motions or explicit kinematic chains.

2.1.4 Task Level Programming : an Implicit Task Description

All the methods discussed thus far have had explicit task descriptions i.e. the order and execution of the task is described in detail by the user within the program. As programmers provide the world model with increasingly accurate and detailed physical knowledge, the incorporation of task planning ability into the work cell becomes possible. Task planners produce task descriptions based on a comparison of the current state of the world model and the requested state.

When presented with a *goal* or final state, a task planner will assemble a solution strategy capable of producing the goal state. The task definition becomes a history of world model states. The resultant strategy is usually composed of *operators* affecting changes on one state to produce another. The development of a strategy requires the generation and evaluation of trial strategies and the selection of the correct strategy based on environmental knowledge.

This approach is significantly more complex than the methods outlined above and represents the ultimate form of user interface: one capable of deducing and performing assembly tasks with little instruction, assistance, or supervision. Unlike other systems where the boundaries between world model data and task description are well established, the task planner's world model is composed of both object level data and rules of operation or heuristics. These rules provide preconditions for the application of an operator to the world state. Realizing this system demands sophisticated modelling techniques [24] beyond those outlined in section 2.1.2 and may include knowledge about kinematics, dynamics, machining or assembly processes. This level of modelling is currently beyond computing power and modelling ability customarily used within the industrial robotics environment but has been implemented on a limited scale in BUILD [2].

2.2 Industrial Off-Line Programming Languages

The limitations of the hand taught robot were seen early on in the 1960's. The first off-line language was MHI developed at MIT [7]. This language allowed one of the first computer controlled robots, MH-1, to move until sensor conditions were fulfilled, a process called 'guarded motion'. Research at Stanford University produced the experimental language WAVE in the early seventies. The ideas developed within WAVE were used to produce one of the first industrial languages for Robots—VAL. Subsequently a large number of research languages have been produced to minimize programming difficulty or optimize robot control. Table 2.1 presents a list of current programming languages available for a variety of robots.

2.2.1 Commercial Languages

In this section some of the representative languages will be used to demonstrate typical robot control language commands. Three languages are used to demonstrate these characteristics: VAL, AL, and PASRO/C. The following is a condensed overview of [5,7,13].

VAL VAL, the most popular robot programming language, is used by Unimation for its series of PUMA robots. VAL has simple control structures such as IF ...THEN, GOTO and FOR ...NEXT loops. VAL lacks traditional structured programming facilities such as parameter passing within subroutines and requires teach pendant data in an off-line/on-line development loop. Since VAL is an industrial language, it is rugged and reliable. VAL-II has corrected much of VAL's earlier shortcomings.

Table 2.1: Currently Available Robot Control Languages

Language	Date	Institution of Origin	Status
AL	1974	Stanford University, U.S.A.	Industrial
AML	1977	IBM, U.S.A.	Industrial
AUTOPASS	1976	IBM, U.S.A.	Research
HELP	1982	Digital Electronic Automation	Industrial
MAPLE	1975	IBM, U.S.A.	Research
PASRO	1983	University of Karlsruhe, F.R.G.	Industrial
PAL	1978	Purdue University, U.S.A.	Research
RCCL	1986	Purdue University, U.S.A.	Research
RAIL	1981	Automatix, U.S.A.	Industrial
RAPT	1979	University of Edinburgh, U.K.	Research
ROBEX	1980	Technical College(RWTH), F.R.G.	Industrial
SIGLA	1978	Olivetti, Italy.	Industrial
SRL	1983	University of Karlsruhe, F.R.G.	Research
VAL	1975	Vicarm (Unimation), U.S.A.	Industrial
VAL-II	1982	Unimation, U.S.A.	Industrial

PASRO/C PASRO/C or simply PASRO is an example of library extensions to existing programming languages, in this case PASCAL and C. Therefore, the language possesses all the advantages of these host languages. Utility functions for world model construction and manipulator commands simplify the programming process.

AL AL and its cousin SRL are representative of the highest level commercial language available. It possesses a syntax similar to PASCAL type languages. In addition a sophisticated world model handler monitors the state of the world through an affixment procedure. This procedure provides AL with an Object Level capacity that allows the user to issue assembly related commands, entirely avoiding robot level instructions.

Most of the remaining commercial languages possess characteristics shown within at least one of the three representatives exemplifying robot programming languages.

Data Structures and Flow Control

Data structures are arguably as important to robot programming as the functions available within a programming language.

Since the introduction of early languages such as FORTRAN, BASIC, and PASCAL, INTEGERs, REALs, and CHARACTERs have become standard data types. While STRINGs and ARRAYs are also standard their representation varies from language to language.

Robotics, however, uses certain combinations of these types often enough to warrant the creation of new data types in some languages. In particular the following types are commonly found:

vector for the description of a vector in three-space.

rotation for the description of an orientation in three-space.

frame for the description of position and orientation in three- space.

jointvector for the description of a manipulator position in joint space.

The roles of record and array structures found in PASCAL are performed by the following constructs:

array identical to the PASCAL, FORTRAN or C array.

record identical to the PASCAL record or C struct.

aggregate unique to AML, this construct is indexed like an array but allows storage of multiple data types in a single aggregate.

Efforts have been made to standardize data types in Robot Control Languages. IRDATA (Industrial Robot DATA) [14], developed in Germany in the early 1980's, was based on the earlier attempt to standardize NC machine tools CLDATA.

IRDATA outlines a set of standards for software/machine interface and data types for industrial machine tools. A high level robot language is used to develop IRDATA robot independent code. This code is further converted by robot device drivers into the appropriate instructions for the given manipulator. This convention enables IRDATA code to be employed on virtually any robot equipped with IRDATA device drivers. The IRDATA standard types are outlined in table 2.2. Blume and Jakob describe the implementation of an IRDATA based system [14].

Flow control such as GOTO and IF ...THEN are common to most robot programming languages. A summary of additional Flow control available for some of

Table 2.2: The IRDATA standard data types condensed from Blume and Jakob [14] *Programming Languages for Industrial Robots.*

BOOLEAN	True=1 of False=0.
INTEGER	an integer number (±2147483647).
REAL	a real number (±1.7 E 38).
VECTOR	three REALs describing cartesian coordinates.
ORIENTATION	three REALs describing orientation.
ADDITIONAL AXIS	a joint angle.
WORLD	a VECTOR and ORIENTATION pair.
JOINT	a data type composed of up to 31 REALs.
CHARACTER	an ASCII character from 0 . . . 255.
STRING	an array of characters
POINTER	memory address (unsigned integer)

Table 2.3: A comparison of current robot control language flow control.

Language	goto	if/then	while/do	for/do	do/until	case	subroutines
AL		●	●	●	●	●	●
AML	●	●	●	●	●		●
AUTOPASS	●	●	●	●	●	●	●
HELP	●	●	●				●
MAPLE	●	●	●	●	●	●	●
PASRO	●	●	●	●	●	●	●
PAL				●			
RCCL	●	●	●	●	●	●	●
RAIL		●	●	●	●	●	●
ROBEX	●	●		●			●
SIGLA	●	●					●
SRL		●	●	●	●	●	●
VAL-II	●	●	●	●	●	●	●

the languages appears in table 2.3. Structured programming is also possible in robot languages but is often less sophisticated than typical implementations of PASCAL or FORTRAN.

World Model Instructions

Typical examples of industrial level world models are those developed for VAL and PASRO. VAL world models are built through a teach in procedure that allows the user to specify frames. PASRO has no innate off line modelling ability beyond traditional PASCAL or C. AML does possess the facilities for the construction of world models through the aggregate data structure. Like PASRO, however, the implementation is left to the user.

Only one industrial language possesses object level world models: AL. As mentioned earlier the world model must be maintained with some mechanism for the creation and destruction of part attachments. These attachment mechanisms are embodied in the `AFFIX` and `UNFIX` commands. By using the qualifier `RIGIDLY`, mutual attachment between parts can be specified.

```
AL    : AFFIX gripp TO box RIGIDLY
```

Absolute and Relative Joint Level Instructions

Most robot control languages come with absolute joint level motion commands that move the robot to a specific configuration. Examples that drive the second joint to 110 degrees:

```
VAL   :     not applicable
AL    : DRIVE JOINT (2) OF ARM1 TO 110
PASRO : drive(2,110)
```

note that PASRO calls are always in small case. Relative motion or the movement of a joint relative to the current position can also be exemplified by:

```
VAL   : DRIVE 2, 20, 100
AL    : DRIVE JOINT (2) OF ARM1 BY 20
PASRO : drive(2,robotjoints[2]+20)
```

Note that VAL only uses relative motion for joint level instructions. They must appear with some speed command (in this case: 100).

Absolute and Relative Robot Level Instructions

Robot level instructions are in terms of cartesian coordinates and are typically represented as a position vector and a RPY or Euler angle vector. For example: to move the gripper to x, y, z (centimetres) and R, P, Y(degrees) $=\langle 50, 10, 12\rangle, \langle 0, 45, 0\rangle$:

```
VAL   : POINT P1
        50,10,12,0,45,0  (from a teach in procedure)
        MOVE P1
AL    : MOVE ARM TO FRAME(ROT(YHAT,45),VECTOR(50,10,12)*CM);
PASRO : SETFRAME(goal,50,10,12,yaxis,45);
        SMOVE(goal);
```

Note that VAL requires a teach in procedure for explicit frame entry.

Relative motion in the x direction of 40 cm and 30 cm in the z- direction:

```
VAL   : DRAW 40,,30
AL    : MOVE ARM1 TO @ + VECTOR(40,0,30)*CM;
PASRO : makevector(relvect,40,0,30);
        frametrans(goal,robotframe,relvect);
        smove(goal);
```

Robot Motion with Velocity, Acceleration, and Duration

Many languages have facilities for monitoring the velocity of the end effector:

```
VAL   : SPEED 200
AL    : MOVE ARM TO targetframe WITH SPEEDFACTOR = 3;
PASRO : SPEEDFACTOR = 3;
        smove(goal);
```

Note that VAL uses a percent measure of normal velocity where the value 200 is equivalent to 200% or twice the normal velocity. The **SPEEDFACTOR** used in AL and PASRO languages is the ratio of the maximum to the desired velocity. In this case one third of the maximum velocity will be applied to the end effector for the move in AL and for all subsequent moves in PASRO.

Acceleration is rarely specifiable within industrial languages. AML uses the following expression for an acceleration of 0.5 cm/s^2:

```
AML   : ACCEL(0.5)
```

Duration is another less common command, appearing only in AL. A typical command specifying a duration of 5 seconds for a movement:

```
AL    : MOVE ARM TO targetframe WITH DURATION = 5 * SEC;
```

Path Control and Intermediate Frames

Point to point manoeuvres from some startframe to a goalframe can be specified by:

```
VAL   :    not applicable
SRL   : PTPMOVE arm TO goalframe;
PASRO : pmove(goalframe);
```

Joint interpolated motion:

```
VAL   : MOVE frame.
SRL   : SYNMOVE arm TO frame;
PASRO : jmove(frame);
```

Cartesian motion:

```
VAL   : MOVES frame
SRL   : SMOVE arm TO frame;
PASRO : smove(frame);
```

Intermediate frames or frames that must be navigated while performing a complex manoeuvre can be specified in a number of languages. For a manoeuvre with two intermediate frames between the start and target positions:

```
VAL   : ENABLE CP
        MOVES interframe1
        MOVES interframe1
        MOVES targetframe
        DISABLE CP
AL    : MOVE ARM TO targetframe VIA interframe1, interframe2;
HELP  : SMOVE(1,#1,xz1,#2,yz1,#3,zz1,#4,rz1,#5,pz1,#6,yawz1,#7)
        SMOVE(1,#1,xz2,#2,yz2,#3,zz2,#4,rz2,#5,pz2,#6,yawz2,#7)
        MOVE(1,#1,x,#2,y,#3,z,#4,r,#5,p,#6,yaw,#7)
```

Note that in VAL the CP switch denotes continuous path motion. This usually results in smooth transitions from motion to motion, in contrast to point to point control which generates discrete path segments, often with discontinuities in end effector movement. In HELP, the move command is composed of a difficult $\langle x, y, z,$roll,pitch,yaw$\rangle$ type notation sent to switches 1 through 7.

Motion with Sensor Monitoring

Sensors can be employed in two roles. The first method uses the sensor as a boolean trigger during the execution of a subsequent operation. For example, a force sensor triggered in the z-direction by the weight of a 100 gram mass:

```
AL     : MOVE ARM2 TO targetframe
         ON FORCE(ZHAT)>= 100 * GM
         DO STOP &;
```

A sensor can also act as part of a control system, monitoring values during an operation e.g. applying a constant force of 700 grams along the z-direction of a drill shaft:

```
AL     : MOVE drill TO below
         WITH FORCE = 700 * GM ALONG ZHAT OF drilltip IN HAND
         WITH FORCE = 0 * GM ALONG YHAT
         WITH FORCE = 0 * GM ALONG XHAT;
```

Note that in this example the force is measured relative to the drilltip.

End Effector Instructions

Absolute gripper instructions usually take the form:

```
VAL    : OPEN
         MOVE gripframe
         CLOSEI
AL     : OPEN HAND TO 5 * CM;
         CLOSE HAND TO 5 * CM;
PASRO  : gripwidth(5)
```

The VAL example demonstrates a delayed gripper operation with the `OPEN` command, executed only during the move. Immediate gripper action is enabled with the 'I' suffix as in the `CLOSEI` directive. The PASRO **`gripwidth()`** command uses either an open or close operation to achieve the specified jaw opening.

Relative gripper instructions are available in AL and similarly in SRL:

```
AL    : CLOSE HAND BY 6 * CM;
```

Velocity, duration, and force sensing can also be specified in the SRL and AL end effector instructions. VAL has boolean gripper force sensing abilities incorporated in a **grasp** command.

Sensor Instructions for Vision Systems

The most common form of vision system is the inspection vision system. These tend to return pass or fail information while collecting part quality data in a well defined work area. Assembly vision systems must both quickly identify and locate objects [14,48]. All assembly vision systems have three characteristics in common.

1. Parts are associated with symbolic strings.
2. Part Position and Orientation data is presented in cartesian coordinates.
3. the system is taught new part characteristics through a "showing" procedure.

Part identification is based on derived image quantities such as object area, number of holes, and largest and smallest radii [14,48,45]. Two industrial systems are worth noting, VALIIV, an extension of VALII, and RAIL, a integrated vision/robot control language.

Part location is performed through a variety of schemes [45] though moments of area [47] (to be described later) are the most popular for 2-dimensional vision systems. The returned location values usually reflect the 2 dimensional visual field, typically resulting in cartesian centroidal x and y values (z is unknown) and a part orientation angle.

VALIIV employs two vision system commands to acquire pictures and locate objects.

```
VALIIV: VPICTURE
        VLOCATE object, 150
```

Where `object` is the returned coordinate frame composed of a derived centroidal position at a known distance from the camera, and a yaw angle.

RAIL is a highly developed system that returns some 45 features of the current image or blob through the function `OBJ_FEAT`. These features can be accessed directly by using the feature variable names. Switches are used to enable and disable the recognition criteria and camera parameters.

```
RAIL   : OBJFEAT(4)
         IF OBJ_NHOLES ==3 THEN
              part = 1
         ELSE
         BEGIN
              magazine = 1
              part = 2
         END
```

Where the quantity 4 in `OBJFEAT` instructs RAIL to compute the number of object holes and the feature `OBJ_NHOLES` is the quantity returned.

2.2.2 Research Robot Control Languages

PAL, RCCL

Since robot motion can be described as a series of kinematic chain conjunctions, tasks can be described by a list of task equations. PAL [11] uses these equations as the basis of an assembly language. Developed at Purdue University, this language experiments with a model based approach to robot programming where typical assembly operations are described by a task equation data set. These task equations are solved prior to the execution of the program during an interactive phase with the user. Unknown frames are defined through a teach procedure similar to VAL and computed by a powerful task equation solver. The program itself consists of a list of MOVE calls to positions in the work volume. This modelling method has been successfully implemented under RCCL [12] a C programming language extension.

RAPT

RAPT is an interpreter based on the APT Numerical Control language and was developed at the University of Edinburgh throughout the seventies by Ambler et al [23,24,25,26]. Object features are first described by planes, holes and surfaces within a CAD system. Object positions are inferred from the conjunctions of objects' features in the workspace. Any single relation between the features of two objects can be described by using at least one of three uniquely defined relation keywords:*against, fits, and coplanar.* Each of these keywords describes relationships between the following classes of *bodies*: a plane *face*, a cylindrical *shaft* or *hole*, an *edge*, and a *vertex*. Specific definitions govern the keywords' algebraic meaning between any two bodies.

While the system would normally develop unique solutions to task equations based on complete relational descriptions, the system is also capable of taking underspecified descriptions of object interrelationships and producing a placement that satisfies that description. An underspecified relation is one that possesses any degree of freedom or potential free movement. The world models discussed thus far have always assumed a single object position and orientation. Often, however, specification of certain quantities is unnecessary for the completion of the assembly.

For example, the positioning of a peg in a hole, expressed in RAPT as

```
fits/shaft of peg, hole of block;
against/bottom of peg, bottom of hole;
```

may be determined by the requirements that the peg's bottom surface is *against* the hole bottom and that the peg is aligned or *fits* in the hole. The specification of roll about the peg axis along the shaft is unnecessary for the assembly to be complete. RAPT generates a solution that will fix the specified degrees of freedom (in this case: three translational and two rotational degrees of freedom) while realizing that the remaining translations or rotations are free (the rotation about the peg shaft).

Significantly, RAPT syntax is a natural extension of the intuitively simple English-like approach to robot programming seen in languages such as AL or SRL. Though it lacks the extensive control structures used in traditional languages and the output must be manually translated and adapted for use by a real world robot, RAPT removes all of the responsibility of world model maintenance from the programmer during the programming phase and tolerates the underspecification of object placement that abounds in typical component assembly by humans. Further development of this approach continues and may lead to a unified method of assembly modelling [27].

AUTOPASS

In the field of general mechanical assembly, a number of shortcomings are immediately evident in robot programming. The complexity of modelling real world 3 dimensional spaces, the heightened attention to detail required by robotic computing (unlike most forms of scientific or business computing), and the use of sensory feedback to complete assembly operations all conspire to make assembly programming impractical for complicated construction projects [8]. An effective language for assembly must solve these problems in a manner that makes robot programming a natural, assembly oriented procedure. This can be accomplished by removing or hiding robot oriented utilities, such as modelling, manipulator, and sensor commands, from the programming environment. Furthermore the assembly should be expressed in a familiar, meaningful manner to the user. The resulting program should reflect the assembly process, not a series of abstract robot actions.

AUTOPASS or AUTOmated Parts ASsembly System , developed by Lieberman and Wesley [8] at IBM in the mid seventies, was designed to explore assembly programming expressed in human terms. While a natural language interface was considered, at that time it was decided that such an approach would be ambiguous and impractical. A middle ground English-like syntax was chosen that would retain the legibility of written English while retaining a precise set of semantic conventions.

The semantic rules confined assembly variables to object names within standard PL/I data types (typically records). Further natural language features included qualifiers such as `WITH` or `UNTIL`, to modify the behaviour of certain commands. An example template from Lieberman and Wesley:

```
PLACE bracket IN fixture SUCH THAT bracket.bottom
```

```
CONTACTS car-ret-tab-nut.top AND bracket.hole
IS ALIGNED WITH fixture.nest
```

Where the capitalized symbols are AUTOPASS keywords. The above example demonstrates the world modelling abilities of AUTOPASS. The system devised model based grasp positions (if not specified by the user) and collision free paths, both requiring motion planning facilities indicative of a limited Task Level programming ability.

The system employed a graph world model similar to the method outlined in section 2.1.3. Each node of the graph was either an object, object component or assembly. The arcs of the graph represent relations of four different types: part-of, attachment, constraint, and assembly component. Unlike previously discussed models, a geometric design processor was used to build the world model in which objects were composed of polyhedrals and primitive volumes. Therefore, a symbol or variable within AUTOPASS could be used to represent any object, surface, edge, or vertex generated by the design processor. A significant advantage of this method was that collision avoidance algorithms could be employed on a realistic level. Since traditional coordinate frame models fail to describe the full extent of the solid and define only points significant to the assembly, they are poor for realistic collision avoidance behaviour. The polyhedral modelling system partially corrects this fault by modelling the entire solid. Curved surfaces could not be directly modelled by the AUTOPASS system, however, so an optional accuracy parameter specifying the degree of approximation was employed for complex surface description.

AUTOPASS commands are divided into the following four types, briefly:

State Change used to move objects about the world model, resulting in world model changes.

Tool Statements used to enable some tool operation. Tool statements tended to be device dependent and were composed for each tool.

Fastener Statements used to specify fastening operations. Like the Tool statements, Fastener statements were device dependent and composed for each fastener.

Miscellaneous Statements declarations of manipulator names, characteristics and spatial features, geometric variable assignments, and assertions of assembly and attachment relationships.

Much of the data and control structures are similar to PL/I, the host language of AUTOPASS. Bonner and Shin [4] point out that while AUTOPASS was easy to use, the English-like appearance of the language was lost when subroutines were called, and that a mainframe was the required host environment.

2.2.3 Comparison of Modern Robot Programming Strategies

It is plain that the quantity and diversity of robot control languages is almost as great as the number of robots. However, certain characteristics are evident amongst these languages and are discussed below.

Robot Level languages, characterized by VAL, possess a limited syntax and are easy to use over simple motions. However, for assembly programming they tend to be difficult to read and write, obscuring the task with robot oriented variables and procedures. Though vision for VAL is available through VALIIV, VAL is essentially dependent on world model information described through the teach pendant.

PASRO solves the bulk of problems associated with readability and programmability. Since world model development is left to the user, it is difficult to comment on the ease of assembly programming in PASRO. Assuming a world model similar to AL's this would be a powerful Object Level language. As it stands, however, the language remains an advanced Robot Level language. PAL and RCCL, while possessing homogeneous transform models and powerful equation solvers, are difficult to read and program for those uninitiated to transform algebra. Unfortunately, PAL does not support subroutines. Perhaps the greatest advantage of PASRO and RCCL is that the familiar host languages, PASCAL and C, offer extensibility, flexibility, and some degree of portability between computer systems.

Object Level languages, while far easier to read and program than Robot Level languages, require significantly greater modelling ability. Languages such as AL and AUTOPASS possess English-like syntax because of these abilities. AL, still adhering to traditional programming methods, allows the user to engage in robot or object level programming, while supporting structured programming philosophies and legibility. AUTOPASS is less traditional in this regard and is limited to higher level instructions. Unfortunately, the AUTOPASS English-like syntax does not survive the transition to structured programming techniques.

Task Level languages tend to be model dependent, since planning decisions must be based on the world model. Perhaps for this reason few task planning systems have been implemented.

An example of a Task Level system, BUILD [2], attempted to address assembly problems such as path planning, assembly stability and modelling. Fahlman, while using a homogeneous transform model, noted that a major weakness in task planning systems was world modelling. Of the time required to write BUILD software Fahlman admits about 80% was devoted to world model construction, despite his efforts to replace model features with heuristics. Fahlman recommended that world modelling for path planning and assembly must be further investigated for assembly programming.

A number of studies have been performed, notably Lozano–Perez [7], Bonner and Shin [4], Blume and Jakob [14], and Gruver et al [5], comparing the various attributes of robot programming languages. These studies compare the language elements and technical detail such as data structures, control flow, path planning, and world modelling ability of the bulk of robot control languages. Lozano–Perez itemized some required characteristics of evolving robot languages: increased sensor integration, better object models, flexible trajectory specification, parallel task execution, and that

> The evidence seems to point to the conclusion that a robot language should be a *superset*[sic] of an established language, not a subset.

Bonner and Shin go further and attempt to compare features such as the structure and complexity of programming, source code readability, and ease of extension within a benchmark palletizing test. An understandable outcome of their comparison is that the AUTOPASS and AL sophisticated modelling systems considerably simplify the programming cycle. Despite Bonner and Shin's warning of possible semantic ambiguities within AUTOPASS, they suggest that some form of English-like syntax should be adopted for readability and understandability in future robot control languages.

The next chapter will investigate the methods and systems developed for natural language understanding and how they have been applied to robot controlled assembly.

Chapter 3

Natural Language Processing and Robot Programming

A number of projects have addressed the possibility of robot programming in English. In particular Winograd's SHRDLU [20], Bock's HIROB [15], Maas and Suppes system [21], and the work of Selfridge et al. [16,17,18] have proven that there is considerable potential for the union of natural language processing and robotics. Before entering a discussion of the difficulties and accomplishments of these projects, it is necessary to suffer a short review of natural language processing methods. This overview should provide the reader with the background needed to appreciate these first attempts at bridging the man/machine communications gap. For further information on natural language understanding methods and techniques see Allen [56], Winograd [20], and Schank and Riesbeck [57].

3.1 Natural Language Processing Methods

Since the field of Natural Language processing is large and complex, a complete description of all of the methods used to automate the comprehension of human language will not be attempted. Instead, a fundamental group of concepts and techniques will be discussed in the hope that the reader will gain a feel for the topic.

3.1.1 What is Natural Language?

Natural Language is a term used to describe the conventions of discourse or exchange of information through text or speech between humans, hence the term *natural*. Though the above definition suggests a rigid linguistic formalism, this is far from the truth. Natural language is composed of a more or less common set of representative symbols or *words*, loosely organized through some *syntax* to express some intended meanings or *semantics*. In essence, the challenge of natural language processing can be divided into two areas:

- deriving the intended meaning from a set of symbols provided by an external agent through a process called natural language *understanding*.
- providing a meaningful symbol set to an external agent from a predetermined meaning through a process called natural language *generation*.

This thesis is chiefly concerned with natural language understanding, though the concepts described here are fundamental to both topics.

Language is then composed of three, not necessarily distinct, elements: a set of symbols, a syntax, and a set of semantic conventions.

3.1.2 Symbols

Any symbol has semantic and syntactic properties. In natural languages, unlike computer languages, a symbol may have multiple semantic and syntactic values. The semantics of a symbol may be a function of either its syntactic placement within an expression or its *context* within a discourse. This results in a limited set of legal, meaningful combinations within a syntax. Further, the construction or morphology of a given symbol must often conform to semantic conventions within the syntax. For example: the addition of suffixes on nouns in English to denote plurality, possession and tense or the choice of pronouns (and articles in romance languages) to denote gender.

3.1.3 Syntax

Though many conventions exist for describing syntax, three fundamental methods commonly employed are

- Regular Grammars
- Context Free Grammars
- Context Sensitive Grammars

These grammars may be described using two possible representations: Backus Naur Form (BNF) notation and Transition Networks.

Backus Naur Form (BNF) Notation

In BNF notation, a grammar is represented by a list of rewrite or *production rules*, with symbols appearing on the left and right hand side of a production symbol, →. The symbols on the left, called *nonterminals*, may be rewritten as the symbols on the right,composed of *terminals* and nonterminals. Nonterminals are symbols that can be expressed by a combination of terminals - – the set of final or most primitive symbols that constitute the language. Since a production rule rewrites the left hand side to a unique right hand side, multiple rewrite rules for a single left hand side are expressed in a shorthand, separating alternatives with an exclusive OR operator, |, in a production rule. A simple sentence grammar in BNF notation appears in figure 3.1. In this grammar the nonterminal symbols appear in upper case, while the nonterminals appear in lower case. The *start symbol*, 'S', is *top* of the grammar since it represents the top of an inverted *parse tree*. The group of terminal symbols is called the *bottom* of the grammar, analogous to the leaves on the inverted tree. This notation will be employed exclusively in this thesis.

Transition Networks

In general, transition networks describe grammars with a set of directed graphs. Each graph is similar to a BNF production rule, linking nonterminal *nodes* together with terminal or nonterminal *arcs*. Once again, nonterminals must be rewritten using the production rules into terminals.

In transition networks a phrase is legal if a graph can be traversed in the network whose terminals match the phrase. The expansion of a transition network involves traversing from one node to another over *labelled* arcs. These labels refer to either other networks, called *push arcs*, or terminal symbols, called *category* or *cat arcs*. When a push arc is traversed or *pushed*, the similarly labelled graph is then expanded. This expansion continues down through network until the expanded portion of the phrase matches traversed cat arcs. The successful completion of a subgraph *pops* the process back to the parent graph for further expansion.

The grammar of figure 3.1 appears in figure 3.2 in Transition Network notation. Note that the BNF grammar above is composed of a series of directed graphs one for each production rule and that recursive arcs form loops.

S → noun VP
VP → verb NP
NP → article NP1
NP1 → adjective NP1 |
noun

Figure 3.1: A Simple Sentence Grammar in BNF notation

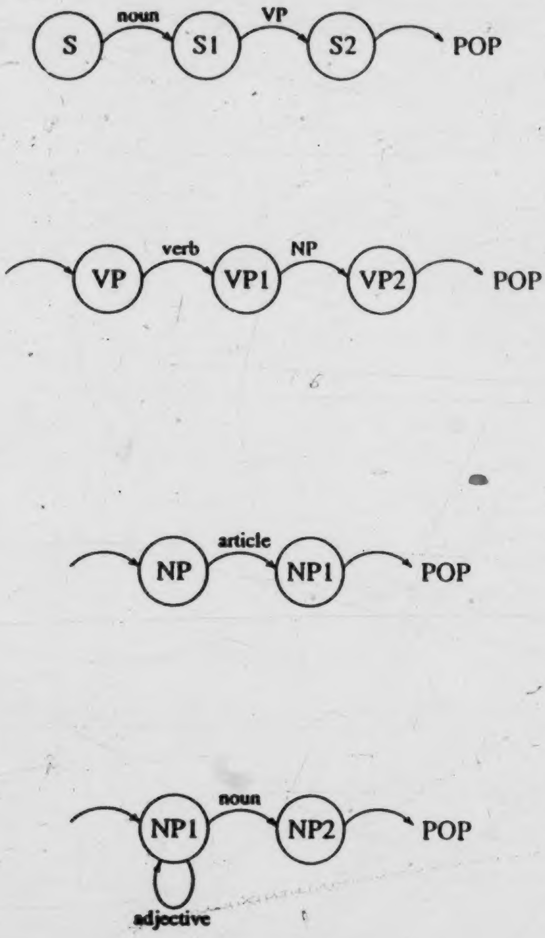

noun
VP
S
S1
S2
POP
verb
NP
VP
VP1
VP2
POP
article
NP
NP1
POP
noun
NP1
NP2
POP
adjective

Figure 3.2: An Transition Network for a Simple Sentence

Regular Grammars and Simple Transition Networks

A regular grammar uses a simple production rule.

$$\langle symbol \rangle_m \quad \rightarrow \quad \text{terminal } \langle symbol \rangle_n \quad m \neq n$$

Where $\langle symbol \rangle_i$ is a unique symbol labelled i. Regular grammars are capable of generating sequences of symbols. For example a Regular Grammar can determine if a string of letters is in alphabetical order. Figure 3.3 shows the grammar in BNF notation while figure 3.4 shows the grammar as a Simple Transition Network. The simple transition network permits cat arcs only. The *generative capacity* or range of languages described by a Regular Grammar is identical to that of the Simple Transition Network.

Context Free Grammars and Recursive Transition Networks

A context free grammar takes the form:

$$\langle symbol \rangle \quad \rightarrow \quad \langle symbol \rangle_1 \ldots \langle symbol \rangle_n \quad n \geq 1.$$

Context Free Grammars (CFG) are a superset of Regular grammars and are able to describe other grammars in addition to Regular Grammars. For example: a CFG can describe a language in which any number of **a**'s are followed by an equal number of b's as in **aabb** or **aaaabbbb** see figures 3.5 and 3.6

A Recursive Transition Network (RTN) is identical to the STN, but also allows the use of push arcs to other networks, including itself, hence the term *recursive.* CFGs and RTNs have identical generative capacity and are often used to automate small subsets of natural language because of their ease of implementation.

S → a S1 |
S1
S1 → b S2 |
S2
S2 → c S3 |
S3
S3 → d S4 |
S4
S4 → e S5 |
S5

Figure 3.3: A Regular grammar generating alphabetical combinations of *a*, *b*, *c*, *d* and *e*

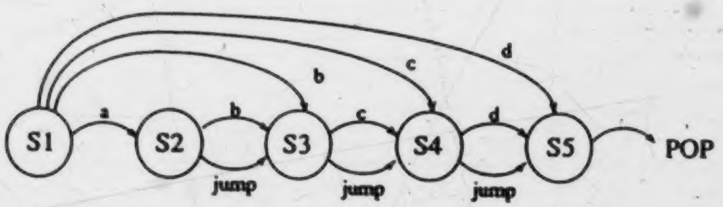

d
c
b
a
b
c
d
S1
S2
S3
S4
S5
POP
jump
jump
jump

Figure 3.4: An STN generating alphabetical combinations of *a*, *b*, *c*, *d* and *e*

S → a b |
a S b

Figure 3.5: A CFG producing equal numbers of *a* and *b*

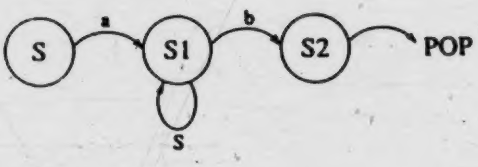

S
a
S1
b
S2
POP
S

Figure 3.6: An RTN producing equal numbers of *a* and *b*

Context Sensitive Grammars

CFGs are a subset of context sensitive grammars. This grammar allows multiple symbols on the left hand side of the production rule with the stipulation that there are fewer symbols on the left than on the right, formally:

$$\langle symbol\rangle_1 \ldots \langle symbol\rangle_m \quad \rightarrow \quad \langle symbol\rangle_1 \ldots \langle symbol\rangle_n, \quad m \leq n.$$

Regular Grammars are a subset of Context Free Grammars, and Context Free Grammars are a subset of Context Sensitive Grammars[1].

3.1.4 Semantics

Natural language interfaces are usually employed over very limited problem domains. These systems serve a specific purpose within well defined limits of syntax and discourse e.g. SHRDLU's [20] use of nouns for locations and objects. Well chosen limits allow a natural language interface to uniquely interpret input sentences. This interpretation, understood by both user and interface, becomes a semantic convention. It is important to note that while a given convention may be similar to human semantic rules of thumb, they are often not identical. Since human understanding is far greater than that of the machine, it is certain that the interface will adopt a more rigid interpretation than the human equivalent. Achieving the comprehension level of human semantic conventions is the ultimate goal of any natural language interface, but concessions must ultimately be made to the limitations of the machine. Winograd [20] points out three levels of semantics:

- the meaning of each symbol

---

[1] These three grammars form a hierarchy called the Chomsky Hierarchy [50].

- the meaning of symbols within syntactic structures
- the meaning of a sentence in context to the linguistic and real world setting.

3.2 Automating Natural Language Understanding

3.2.1 Divide and Conquer Approaches

The most common, though not necessarily most efficient, method of natural language understanding is to retrieve an expression from the user, verify the syntax through a *parsing* procedure, and, later, derive some meaningful semantic interpretation. The following sections will describe this 'divide and conquer' approach.

Basic Parsing Techniques

Parsing an expression of any language is composed of two related tasks:

1. verifying the legality of the expression by confirming that the organization of the symbols conforms to the syntax
2. producing a structure reflecting the syntax of the expression

Once the sentence has been divided into a list of tokens, the tree-like sentence structure can be *parsed* or constructed from the syntax. Though a given parser's exact behaviour depends on the system chosen to represent the language, CFG and RTN systems usually employ one of three fundamental methods:

Top Down where the expression is parsed by evaluating the grammar from the start symbol down to the terminal symbols,

Bottom Up where the expression is parsed by evaluating the grammar from the terminal symbols up to the start symbol,

Mixed Methods where the expression is parsed by partially evaluating the grammar from the terminal symbols up, and partially from the start symbol down, until agreement between both systems is established.

A top down parser applies the rewrite rules from left to right until a combination of terminals matching the token list is generated. The leftmost nonterminal symbol is rewritten into a new expression using a production rule. This leftmost rewriting continues until a set of terminals that match the head of the token list is produced. If the generated terminals at the bottom of a parse tree branch do not agree with the token list, the parser must use an alternative rewrite rule. A complete failure to generate the token list implies the sentence does not conform to the syntax. This method attempts to find a legal token list for a syntax.

This pursuit of the leftmost symbol to the 'bottom' of the tree is called a *depth first* search. Depth first searches require that the routine retain and pursue alternative rewrite rules should other rules fail. This property, called *backtracking*, allows the routine to explore, if necessary, every rewrite alternative during the parsing of grammar.

Another search method, *breadth first* search, rewrites each symbol using all possible rewrite expressions from left to right, before pursuing the rewrite rules for the leftmost symbol in the new 'daughter' expressions. This requires retaining an ever growing list of possible parse descriptions of the token list.

A bottom up parser attempts to assemble legal syntactic structures based on the token list. Bottom up parsers use the Right Hand Side of a production rule to assemble the Left Hand side. Once again the operation can be depth first or breadth first. This method attempts to find a legal syntax for the token list.

One drawback of a pure top down method is that structures are often rediscovered during a parse. For example, in figure 3.7 the grammar: The sentence: "*The man gave the boy the book.*" will initially be rewritten as a NP VP. The NP will rewrite to the noun phrase, *The man*. The VP will become:

VP → verb DOBJ

where *gave* is the verb. The DOBJ is rewritten to the NP, *the boy*. This is plainly incorrect, since the tokens *the book* are unidentified, so the parser must backtrack to the next VP alternative:

VP → verb IOBJ DOBJ

rematching *gave* to verb and rewriting IOBJ and DOBJ symbols as the NPs: *the boy* and *the book* respectively. It is easily seen that the parser is forced to 'rediscover' the verb, *gave*, and the noun phrases *the boy* and *the book*.

The bottom up method avoids rediscovering *symbols* by using the syntactic property of each token to guide the parse. This means that words with multiple

S → NP VP
VP → verb DOBJ
VP → verb IOBJ DOBJ
IOBJ → NP
DOBJ → NP
NP → art noun

Figure 3.7: A Context Free Grammar for a Transitive English Sentence

syntactic values, such as the word *iron* (with potential syntactic values of adjective, verb, and noun) can produce a variety of unsuccessful syntactic results in a sentence e.g. *Iron irons iron shirts* . Only a few interpretations will prove to be legal and, in this sentence, only one syntax will prevail: ferrous irons press shirts.

The search for the legal parse will, again, require a backtracking or breadthfirst search technique to pursue alternatives.

While the top down method suffers from *symbolic* redescription, the bottom up method suffers from *syntactic* redescription. The mixed method partially addresses these redundancy problems by establishing a set of legal syntactic substructures with a bottom up strategy, typically noun, verb and prepositional phrases. The parser will then determine the final syntax with a top down parse. The bottom up phase results in a 'shallower' search tree for the top down phase. Mixed method parsing can be considerably faster than pure top down or bottom up approaches since sections of the parse tree need not be reinvestigated.

Though some parsers determine only the legality of a sentence, returning a pass or fail judgement on the token list, semantic routines must be able to examine the sentence structure during the semantic analysis. This requires some data structure that reflects the tree–like, self–referential structure of natural language. One such record-like structure used in LISP is the *property list*. This employs a linked list of nested property lists, ultimately describing the syntax. From Maas and Suppes [21]:
Add this number to the number you remembered

```
(IMPERATIVE VP+Advs (TV+NP add (NP+Adjs number this))
                    (PrepPh to
                     (N+rclause (NP+adj number the)
                      (that you remembered THAT&SLOT))))
```

The same sentence in a PROLOG record structure is as follows :

```
imp_vp_advs(
  tv_np(add,
    np_adjs(number,[this]),
      prepph(to,
             n_rclause(np_adjs(number,[the]),
             that_slot(that,[you,remembered])))))
```

A grammar that provides this data structure is called an *augmented grammar*. The CFG and RTN formalisms become Augmented CFGs and Augmented Transition Networks(ATNs) respectively.

Semantic Interpretation

Though not a rule, the data structure returned by an augmented parser is often decomposed top down, from the exterior to the interior structures. The syntax is combined with the semantic properties of the terminal symbols to construct, in a bottom up manner, unique semantic definitions of important syntactic substructures.

One method of classifying the relationship between semantics and these substructures is through type hierarchies that describe the physical world. A type hierarchy classifies objects into tree-like groups and subgroups. For example a possible hierarchy of nouns might be the group **phys_obj**, divided into subgroups describing different physical objects. The book "The Lord of The Rings" might be classified in a pseudo-PROLOG type hierarchy as:

```
phys_obj(inanimate(book(fictional(fantasy(The_Lord_of_The_Rings)))))
```

while a Newfoundland dog might be classified as:

```
phys_obj(animate(dog(working(friendly(Newfoundland)))))
```

Verbs can also be classified in this way. For example verbs can be either *events* (e.g. to laugh) and *states* (e.g. to want, own, or be) [56].

Another method is through *Case Relations*. A summary of a set of cases is condensed from Allen in table 3.1.

3.2.2 Unified Approach

An entirely different approach is to tackle the syntax and semantics simultaneously using a combination of traditional natural language grammar and semantic rules of thumb. This can be typified by three possible methods:

1. Interleaved Parsers
2. Semantic Grammars
3. Conceptual Dependency

The interleaved parser brings semantic rules to bear on the syntax to assist in the parsing procedure. Interleaving may be as simple as determining semantically legal noun phrases or as complex as performing a semantic evaluation after each production rule. Regardless of the influence of the semantics, traditional English syntax retains a major role in determining the validity of a sentence in these parsers.

While the interleaved parser maintains the separation of syntax and semantics, semantic grammars attempt to integrate the properties of English grammar and meaning into a single grammatical representation. Pragmatic production rules, not necessarily adhering to standard English grammar, describe common phrases used in the problem domain. A fragment semantic grammar of the LADDER database query system [19] appears in figure 3.8.

Another approach is to let the meaning of each word guide the parse forward using pure semantics, only resorting to the syntax for isolating difficult word groups.

Table 3.1: A comparison between CD slots and Semantic Roles, based on Allen [56] and Schank and Riesbeck [57]

Case	Sub Cases	Definition	Syntax	CD Equivalent
CAUSAL AGENT		The object that caused the event to happen.		
	AGENT	intentional causation	SUBJECT in active sentences Preposition *by* in passive sentences	Approximately Equivalent to ACTOR
	INSTRUMENT	Force or tool used in causing the event	SUBJECT in active sentences with no AGENT Preposition *with*	
THEME		The thing that was affected by the event	Usually the DIRECT OBJECT	Related to OBJECT and ACTOR
EXPERIENCER		The person who is involved in perception or in a psychological state	SUBJECT in active sentences with no AGENT	
BENEFICIARY		The person for whom some act is done	The INDIRECT OBJECT with transitive verbs Preposition *for*.	
AT	AT-LOC	The state of some dimension where the event occured also called LOCATION	Prepositions *in*, *on*, *etc*	Related to FROM
TO	TO-LOC	Final Location also called DESTINATION	Prepositions *to*, *into*, *etc*	Related to DIRECTION
FROM	FROM-LOC	Original Location also called SOURCE	Prepositions *from*, *out of*, *etc*	Related to DIRECTION

S → what is ATTRIBUTE of SHIP
ATTRIBUTE → the ATTRIB | ATTRIB
ATTRIB → speed | length | draft | beam | type
SHIP → SHIP-NAME | the fastest SHIP2| the biggest SHIP2 | SHIP2
SHIP-NAME → Kennedy | Kitty Hawk | Constellation | ...
SHIP2 → COUNTRYS SHIP3 | SHIP3
SHIP3 → CLASS LOC | CLASS
CLASS → Carrier | Submarine
COUNTRYS → American | French | British | Russian | ...
LOC → in the Mediterranean | in the Pacific | ...

Figure 3.8: A Semantic Grammar fragment from LADDER [19]

One method, called Conceptual Dependency (CD), performs semantic analysis of sentences within a discourse through a series of *concepts*. The underlying premise of CD is that any sentence can be described by a combination of *events*, where an event has the following properties:

ACTOR a concrete object capable of producing actions

ACTION one of eleven primitive actions characterizing an action type i.e. ATRANS, MTRANS, SPEAK, INGEST, PTRANS, MBUILD, GRASP, EXPEL, PROPEL, ATTEND, MOVE.

OBJECT the object acted upon.

DIRECTION the direction the action moves the object.

As each word is analyzed from left to right in a sentence, a semantic description describing the word, called a *request*, is evaluated and the results placed on a *concept list* or C- LIST.

The evaluation of each request, similar to an IF–THEN rule, involves performing a word related *test* and an *action* based on the test's outcome. Each test searches the C-LIST for structures relevant to the current word. A successful test results in the construction of a new semantic structure. If the test fails, the request is placed on a *request list* or R-LIST. The typical evaluation of a word can be described in the following procedure by Birnbaum and Selfridge [57].

1. Read the next lexical item (word or idiomatic phrase) from the input, reading from left to right. If none the process terminates.
2. Load (activate) the requests associated with the new item into the R-LIST.
3. Consider the active requests in the R-LIST.
4. Loop back to Step 1.

An example of a typical evaluation appears in Figure 3.2.

Table 3.2: An Example of Conceptual Analysis on: *Fred gave Sally a book.* condensed from Birnbaum and Selfridge [57].

Next Word	Test	Actions	Consider Requests
Fred (REQ0)	True	add to the C-List: (PP CLASS (HUMAN) NAME (FRED))	true (PP CLASS (HUMAN) NAME (FRED))
gave (REQ1)	True	add to the C-List: (ATRANS ACTOR (NIL) OBJECT (NIL) TO (NIL) FROM (NIL) TIME (PAST)) Activate Requests REQ2, REQ3, REQ4.	true (ATRANS ACTOR (NIL) OBJECT (NIL) TO (NIL) FROM (NIL) TIME (PAST))
REQ2	Does a human precede ATRANS on C-List?	Place human in ACTOR slot	true (ATRANS ACTOR (FRED) OBJECT (NIL) TO (NIL) FROM (NIL) TIME (PAST))
REQ3	Does a human follow ATRANS on C-List?	Place human in TO slot	false
REQ4	Does a physical object follow ATRANS on C-List?	Place object in OBJECT slot	false
Sally (REQ5)	True	add to the C-List: (PP CLASS (HUMAN) NAME (SALLY))	true (PP CLASS (HUMAN) NAME (SALLY))
REQ3	Does a human follow ATRANS on C-List?	Place human in TO slot	true (ATRANS ACTOR (FRED) OBJECT (NIL) TO (SALLY) FROM (NIL) TIME (PAST))
REQ4	Does a physical object follow ATRANS on C-List?	Place object in OBJECT slot	false
a (REQ6)	Is there a new structure on the C-List?	Mark as an Indefinite reference.	false
book	True	Add to C-List: (PP CLASS (PHYSICAL-OBJECT) TYPE (BOOK))	true (PP CLASS (PHYSICAL-OBJECT) TYPE (BOOK))
REQ6	Is there a new structure on the C-List?	Mark as an Indefinite reference.	true (PP CLASS (PHYSICAL-OBJECT) TYPE (BOOK) REF (INDEF))
REQ4	Does a physical object follow ATRANS on C-List?	Place object in OBJECT slot	true HALT
Result	ACTOR (PP CLASS (HUMAN) NAME (FRED)) OBJECT (PP CLASS (PHYSICAL-OBJECT) TYPE (BOOK) REF (INDEF) TO (PP CLASS (HUMAN) NAME (SALLY)) FROM (PP CLASS (HUMAN) NAME (FRED) TIME (PAST)		

3.3 Robot Programming with Natural Language

3.3.1 SHRDLU : Winograd, 1972

Though not designed for robot control, SHRDLU [20], is perhaps the most popular example of a natural language interface. SHRDLU employed a combined syntactic/semantic natural language subset to perform simple construction in a *Blocks World* with an imaginary robot. The system would execute user commands, answer queries about the world, and both plan and justify its activities.

The Natural Language module was composed of seven submodules. An INPUT module performed morphological analysis of the commands before passing them on to a GRAMMAR module that coordinated a PROGRAMMER parser and SEMANTICS semantic processor. Responses were generated by the ANSWER module.

The INPUT module converted the input string from a word list into word classes, a set of root definitions contained in the DICTIONARY and modified as the morphology required. Tokens were reviewed for tense and number agreement (e.g. *He went home.* where *went* is a form of the infinitive *to go* modified into 'past tense').

Winograd's parser was similar to an interleaved ATN processor. Typical ATN processors describe sentence substructures by 'pushing' *registers* onto a stack. Note that a 'pop' for a daughter network becomes 'push' for a parent network. For example, for a noun phrase, the registers `DET`, `ADJ`, and `NOUN` might be pushed onto the stack by *determiner*, *adjective*, and *noun* networks or push arcs and popped into a parent network when the noun phrase network is complete. The structure popped from the noun phrase network and pushed onto a parent network for *the black dog* in LISP would be:

```
(NP (DET the) (ADJ black) (NOUN dog))
```

Unlike traditional ATN parsers, SHRDLU used procedures rather than registers to describe the grammar. These procedures were written in a LISP based interpreter, PROGRAMMAR, and was both top down and depth first. While predominantly context free, the parser also used context sensitive information to check word and phrase agreement by inspecting relevant nouns, verbs, and phrases. The parser was able to modify its backtracking behaviour with 'demon' programs to handle special semantic cases. These demons would alter the depth first execution of a parse in order to avoid nonsensical or ambiguous semantic results. Semantic and syntactic processing was often interleaved by considering the semantics immediately after the evaluation of a legal production rule and was performed in the SEMANTICS module.

This semantic construct was passed to a deductive module written in PLANNER that directed a MOVER to manipulate blocks in the world model.

Winograd deliberately used a simplified world model composed of simple three dimensional shapes. Objects could be placed in, on, and beside one another. Position was a simple vector in three space and, as Winograd [20] comments, was: "... designed less as a realistic simulation of a robot than to give the system a world to talk about." This world was presented in real time on a graphical display.

3.3.2 A High-Level Hierarchical Robot Command Language (HIROB) : Bock, 1984

This system was employed an English syntax for the purpose of robot instruction [15] as opposed to true natural language understanding. An instruction was composed of a three tiered hierarchy including high level English language instructions(HIROB), middle level flow control instructions (MIDROB), and a set of low level robot control directives (LOROB). Any HIROB procedure could be described by MIDROB or

LOROB instructions. HIROB is divided into three modules:

- The Parser/Scanner/Recognizer (PSR),
- The Procedure Management System (PMS),
- The MIDROB Interpreter (MINT).

The Parser/Scanner/Recognizer employed a keyword matching module, the Scanner, a procedure reference file, PROCNAME, and a top down parser composed of a CFG with 36 production rules. The scanner searched for keywords that matched procedure names stored in PROCNAME. A procedure name could be a word or phrase whose syntax corresponded to one of the nonterminals in the CFG. Given the portions identified by the scanner module as terminals, the parser verified the syntax of the sentence. Though Bock does not state so directly, this seems to be a mixed method system, using the scanner to determine legal substructures and verifying the overall sentence syntax with a final top down parse.

The Procedure Management System is a user friendly environment, permitting the creation, alteration, and destruction of HIROB procedures. HELP and LIST utilities are also available.

The MIDROB interpreter converts the HIROB procedures, composed of MIDROB and LOROB directives, into a LOROB execution stream issued over a serial line to the robot.

The robot is taught new procedures through the `CREATE` directive. This invokes a teach environment where the user provides both the syntactic value of the string to be defined and a program written in MIDROB or LOROB BASIC–like commands defining the semantics of the string.

Bock [15] concludes that:

> " Composing in English obviates the need for a programmer, except for the lowest level procedures which use Low-level Robot Command Language (LOROB) statements and MIDROB directives which control the logical flow in each procedure. As the repertoire of LOROB procedures grows, less and less use of the LOROB programmer will be required."

Apparently this system requires the presence of an experienced LOROB programmer in the early stages of system development, somewhat defeating the purpose of the HIROB approach. Though Bock makes no mention of variable representation, treatment or world modelling conventions, the examples provided indicate that the world model is simple and not directed towards assembly. Nevertheless Bock's CFG successfully takes advantage of the fact that an imperative command set describes most robot commands, a property also reflected in the AUTOPASS templates.

3.3.3 A Natural Language Interface for an Instructable Robot : Maas and Suppes,1985

This system employs the narrow domain of addition and subtraction to test the applicability of natural language understanding to robot programming. The world model consists of a mapped grid, each grid element is labelled by a coordinate pair and contains one number. The system is divided into three LISP modules: the parser, the translator, and the interpreter.

Parser

The parser is, once again, based on a CFG description of the imperative command English subset. Maas and Suppes make no claims about their grammar other than a

CFG compiler was used to parse strings based on a CFG provided at runtime. The parser is composed of three stages: a pre-, main, and post-parser.

The preparser formats the command sentence for input to the main parser.

The main parser, called CREATE, compiles a CFG 'grammar file' into a set of tables that rapidly determine all the legal parses of a given sentence without backtracking. The parser produces a list of possible legal parses.

The post parser reviews the legal parses, eliminating bad parses, weighing each one for 'acceptability', providing warnings if none are acceptable, and 'prettyprinting' the successful parse.

Translator

The successful parse is then routed to a *translation table* composed of a *template* or syntax and a *result* or semantic translation. The parse is compared against this list of 'templates' and, when matched, translated into the semantic equivalent.

Each template has a unique syntactic structure. Though parts of the template structure will be fixed, some of the syntactic substructures will be unspecified as spots for *pseudo-* variables.

For example the sentence *Look at the top number.* is parsed as

```
(IMPERATIVE VP+Advs look (PrepPh at (NP+Adjs number the top))).
```

The matching template,

```
(IMPERATIVE VP+Advs look (PrepPh at [NOUN]))
```

where `[NOUN]` matches with `(NP+Adjs number the top)`. Since the template set may contain multiple matching candidates, a failure to bind pseudo-variables in one candidate engages a backtracking algorithm to examine the next one. When a match is verified, a semantic rule is used to convert the parse into a LISP semantic construct.

A typical semantic rule converts the term `(NP+Adjs number the top)` matched with `NOUN` into

```
((NOUN INTERSECT-CUES (OBJECT NUMBER) (LOCATION TOP)))
```

The remaining template is then converted by a similar rule from

```
(IMPERATIVE VP+Advs look (PrepPh at [NOUN]))
```

to `(LOOK-AT [NOUN]))`. The final semantic construct then becomes:

```
(LOOK-AT ((NOUN INTERSECT-CUES (OBJECT NUMBER) (LOCATION TOP))))
```

An interpreter then evaluates the resultant expression *coercing* arguments to numbers (since this is an arithmetic robot). Coercion implies that arguments in the semantic list are operated on by functions to produce numbers. In the example `(LOCATION TOP)` is coerced by a function `COERCE-TO-LOCAT-COORDS`, converting keywords into coordinate offsets. The resulting arithmetic expressions are subsequently evaluated.

While this system has much in common with Bock's system, both employing a CFG and template matching scheme, Maas and Suppes' semantic post processor distinguishes this processor. Since it employs a specialized arithmetic world model, semantic interpretation is straightforward and relatively unambiguous.

Maas and Suppes admit that one failing of the system was that it was not possible to generalize the resultant subroutines, meaning that these subroutines could not take variable arguments.

3.3.4 A Natural Language Interface to a Robot Assembly System : Selfridge et al., 1986

The previous examples made no attempt to address the difficulties of natural language processing within a complex assembly environment. As discussed earlier, assembly programming requires a kinematically descriptive world model. The world model must, therefore, be accessible from the interface in a 'natural' manner, describing the real world as closely as possible.

However, Selfridge et al [16,17,18] have successfully applied an English interface to a work cell performing real world assemblies. Equipped with a stereo vision system, two five degree of freedom robots, and world modelling system, the interface employed conceptual dependency analysis to discover the meaning of an input string.

The natural language module, written in LISP, converted the CD representation of a command into an intermediate task plan code. Each plan was composed of a *goal part* and a *plan part*. Each plan part was in turn the goal part of a subplan. Each subplan is expanded until the original goal part is fulfilled. In this way a single goal was a hierarchical tree of subgoals, the terminals of which were primitive LISP robot and world model activities.

The world model retrieved information from the camera images, deriving centroidal, area, range, circularity, and shape data of objects in the work area. This data was further converted into a three-dimensional semantic representation. The natural language module's output was then interpreted in terms of this representation.

This system was capable of understanding world model queries and could be 'taught' both the shape of new objects through English description and the plans of new tasks through a top down teach phase. The teach module, invoked by the system when an unknown command was encountered, built plans based on a users explanation of the new task. If any unknown subtasks were encountered in the explanation, the teach system would be reinvoked to define the task. In this sense the teach system was hierarchical, requiring task definitions in terms of only previously defined tasks. The system could also generate English statements and responses to queries, again using the CD representation.

The interface greatly simplified the task of robot programming, given the complexity of two robots, a vision system, and a descriptive world model. Being able to both understand and generate English in an assembly oriented dialogue, this system effectively demonstrated the feasibility of natural language interfaces for assembly work cells. Unfortunately, the system relied on a large minicomputer (a VAX 11/780) and a dedicated vision microcomputer (an APPLE II+) to run the software. Further, the CD representation requires the prior definition of a lexicon in LISP hence the relatively small dictionary of approximately 50 words [17]. Further, Vannoy and Selfridge point out that they did not address the issue of subroutine generalization.

3.4 The Bare Essentials of Natural Language Interfaces for Robots

Though the example systems discussed thus far possessed varied parsing methods and different degrees of complexity, each system successfully interpreted English commands. Both Winograd's and Selfridge's systems were subtle semantic/syntactic hybrids while Maas and Suppes' system was less complex, operating in a narrower domain with separate syntactic and semantic modules. All three systems, were designed to *understand* user commands with semantic constructs. Bock relied solely on a template matching method and made no attempt to produce semantic constructs. Though these systems have different capabilities and world models, they do have similar surface behaviour i.e. *They allow the user to instruct a 'robot' with natural language commands.*

Natural language understanding presupposes a semantic element in the processing software. In the case of traditional CFG or ATN parsing methods there is often a distinct division between the syntax and semantics. Efficient parsing methods have been developed for CFG and ATN systems and, in the case of CREATE, are often orchestrated by machine. This leaves the system developer to concentrate on a set of semantic formalisms based on syntactic information and requirements of the problem domain. Unfortunately this separation of semantics increases the processing time and size of the NLI, reducing the ability to generate flexible semantic constructs.

Alternatively, the CD parsing method removes the barrier between syntax and semantics, reflecting human language comprehension methods. Unfortunately, the implementation of a CD processor is labour intensive, given that each word in a CD dictionary must be characterized by a unique LISP construct.

If such a wide variation in semantic techniques produces the same surface behaviour, one cannot help but ask:
How much semantic processing is required for a work cell NLI?
A review of earlier discussions outlines some key Work Cell NLI unification attributes:

- a manipulator and sensor array to change and monitor the environment,
- a *hidden* world modelling system for the simulation of assemblies during construction,
- a set of semantic formalisms describing both world model and task definitions to the natural language processor,
- a natural language processor employing a subset of English commonly used in assembly.

This clearly shows that some semantic constructs for world model representation are necessary. However, the example systems also show that the adoption of semantic conventions within the language subset can reduce semantic processing e.g. in SHRDLU the restriction of nouns to locations and objects in the work cell.

Given that a work cell NLI need not *fully* comprehend English semantics, can work cell NLIs be designed to function on the scale of a personal computer? To investigate this potential, an experimental NLI has been designed and incorporated into a vision equipped assembly work cell hosted on an IBM-PC.

Chapter 4

The Experimental Work Cell

The following sections will discuss the hardware and software support for the University of Alberta Work Cell. A schematic of this work cell appears in figure 4.1

4.1 The Manipulator

The work cell manipulator is a 6 degree of freedom RSI Excalibur articulated robot with a separate controller driving 6 servomotors. The robot has a published accuracy of approximately ±1.3mm, a reach of about 1 metre, and can lift about 4 kilograms.

Excalibur can be taught manually with the use of a *master* arm, a scale model of the manipulator, or off line from a personal computer, in this case an IBM compatible with 16MHz 80386 microprocessor, 80 MB Hard drive and 2MB RAM.

The controller understands 35 possible commands, the most common of which are the `MOVETO` command, sending the robot 6 specified joint angles, the `OPEN` and `CLOSE` commands for gripper control, and the `MANIP` command, returning the manipulator's joint positions. Formally:

```
MOVETO joint1 joint2 joint3 joint4 joint5 joint6 gripper
MANIP
OPEN size
CLOSE size
```

Though the robot comes with an off line language, RCL, the language lacks advanced control and data structure capabilities.

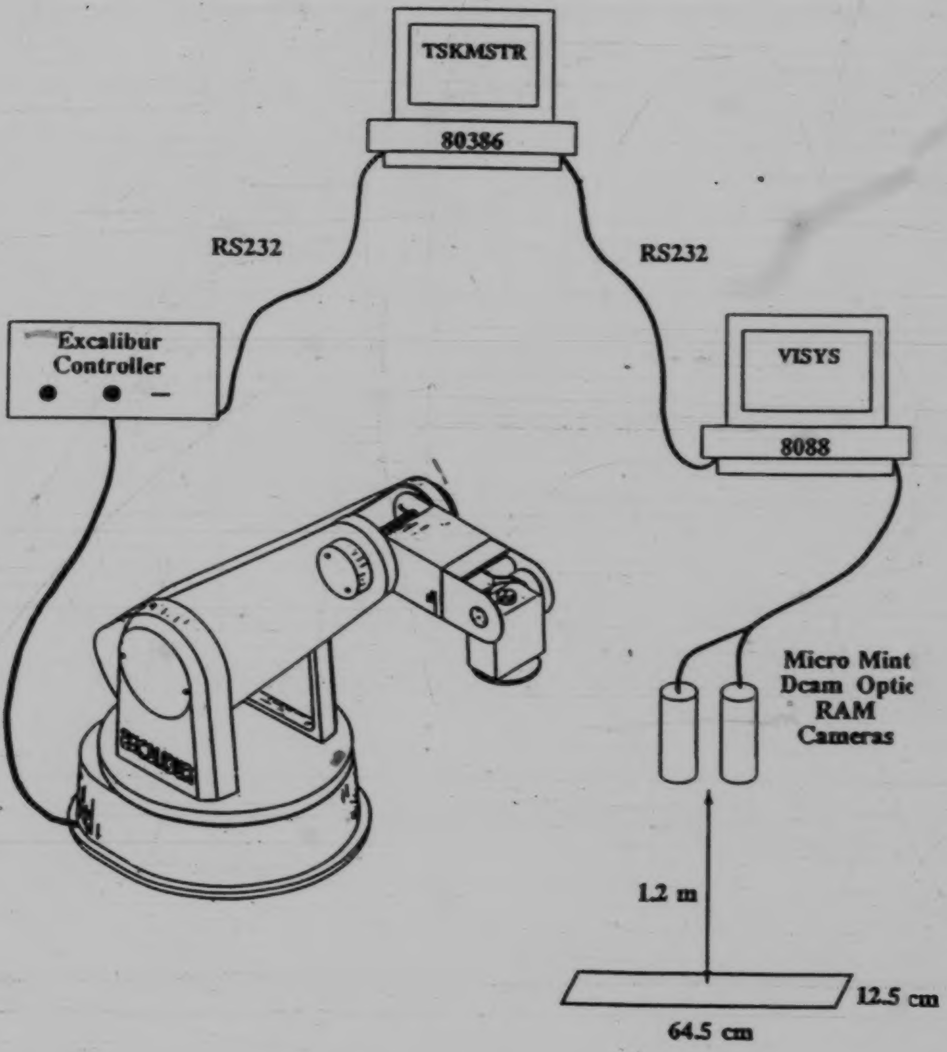

TSKMSTR
80386
RS232
RS232
Excalibur Controller
VISYS
8088
Micro Mint Dcam Optic RAM Cameras
1.2 m
12.5 cm
64.5 cm

Figure 4.1: A schematic of the University of Alberta's prototype work cell.

4.2 The Vision System

4.2.1 The System's Physical Attributes

The work cell has two Micromint D-cam Optic RAM cameras, mounted 1.2 metres perpendicularly above the work area. Each camera has a rectangular 128 × 256 pixel sensor array. Since the cameras are sensitive to the infrared, two 100 watt incandescent light sources are available for consistent lighting. At 1.2 metres the view area is relatively limited at 12.5 by 64.5 centimetres using a 16mm lens.

The cameras are controlled remotely from an inboard vision processing card installed in an IBM PC with a 4.77 MHz 8088 microprocessor, 20 MB hard drive and 640K RAM. The vision hardware can be operated from any MS-DOS language. Though the board is capable of generating grey scale images, the implemented system produces binary silhouettes of objects in the view area.

Unfortunately, vision hardware limitations forced the camera to remain isolated on the 8088, preventing installation on the faster 80386. Though this does not alter the intended operation of the work cell, the work cell does suffer a penalty of increased vision processing time.

4.2.2 Vision System Operation

The vision system appears to the user as a menu driven program with three operational alternatives: the Local Mode, the Shape Editor, and the Remote Mode. The reader is directed to the flow chart in figure 4.2.

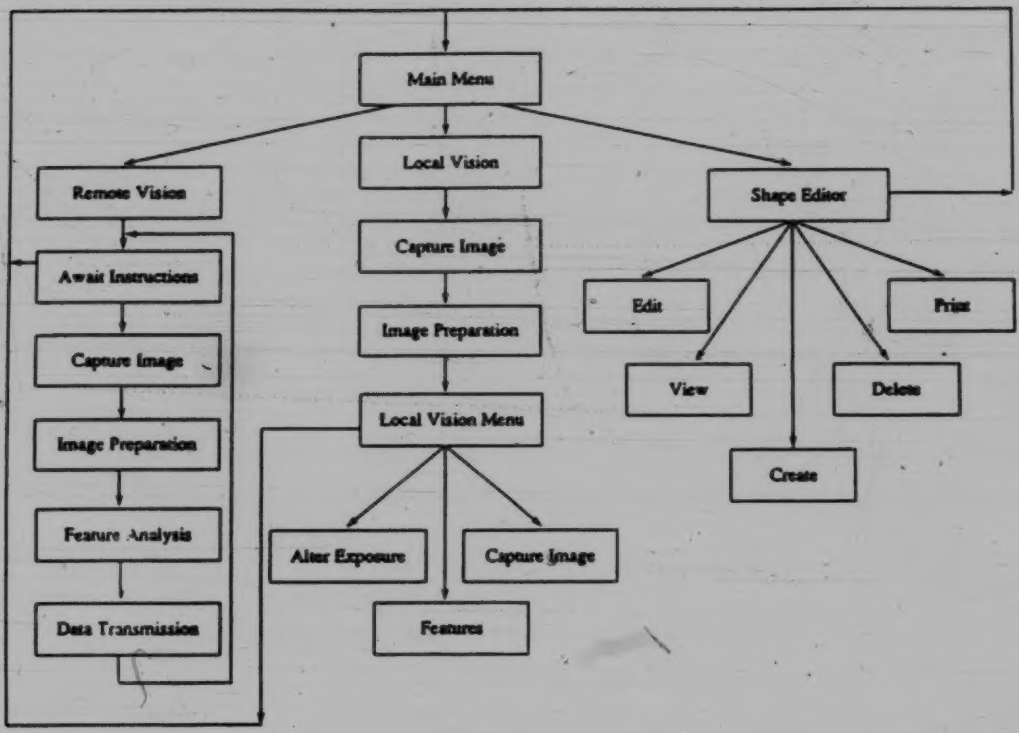

Main Menu
Remote Vision
Local Vision
Shape Editor
Await Instructions
Capture Image
Image Preparation
Feature Analysis
Data Transmission
Capture Image
Image Preparation
Local Vision Menu
Alter Exposure
Capture Image
Features
Edit
Print
View
Delete
Create

Figure 4.2: The VISYS binary vision system flow diagram.

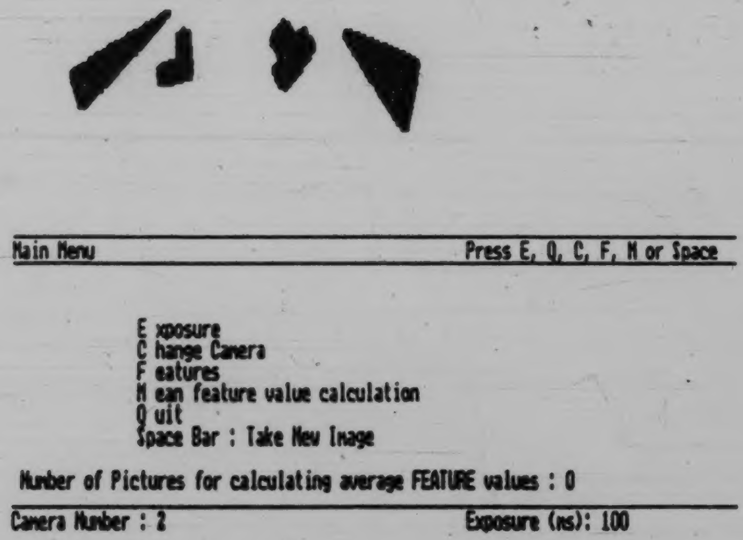
The Local Mode

The Local mode of operation is used to determine picture quality, heavily dependent on environmental lighting, and to introduce the vision system to new objects in a teach cycle.

As mentioned above the Optic RAMs are sensitive to the infrared. This means that ambient lighting and room temperature will play a major role in camera exposure times. Short exposures are desired to both shorten the vision processing cycle and limit image contamination due to infrared noise. To reduce exposure times, increase image contrast, and reduce the influence of ambient conditions, two incandescent light sources are available to light the work space. The optimum exposure can then be determined in the Local Mode (typically 100 milliseconds).

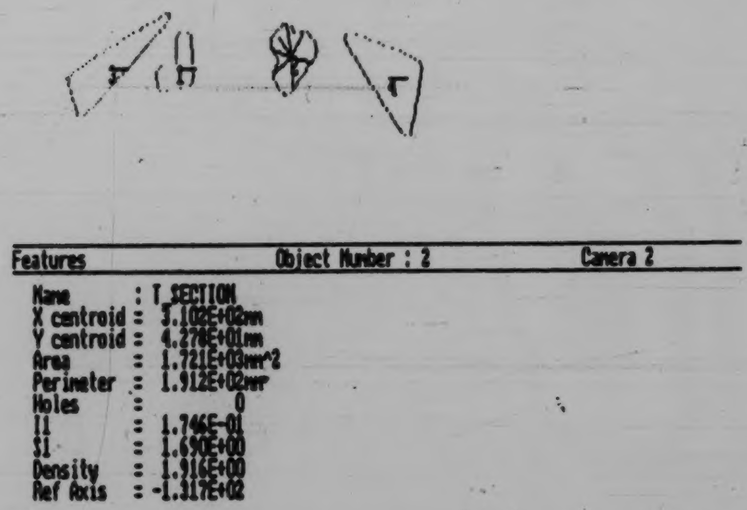
To teach the vision system new silhouettes, new objects are placed in the camera's field of view as in figure 4.3. The captured image is presented on the monitor and, if deemed satisfactory by the user, evaluated for Object connectivity and identification shape descriptors (figure 4.4). The computed values returned from this routine, also presented on the monitor, may be statistically evaluated on request. The results of the evaluation can be used to construct the identification routine's object library data during the editing phase. Object identification ranges, either determined from trial and error or from the statistical evaluation, are then incorporated into *Library Object* records in the shape editor as in figure 4.5. Object records appear as:

```
type
   ObjectName = string[20];
   Range = array[1..2] of Real;
   LibraryObjects = record
                          Name : ObjectName;
                          Holes : Integer;
                          Area,
```



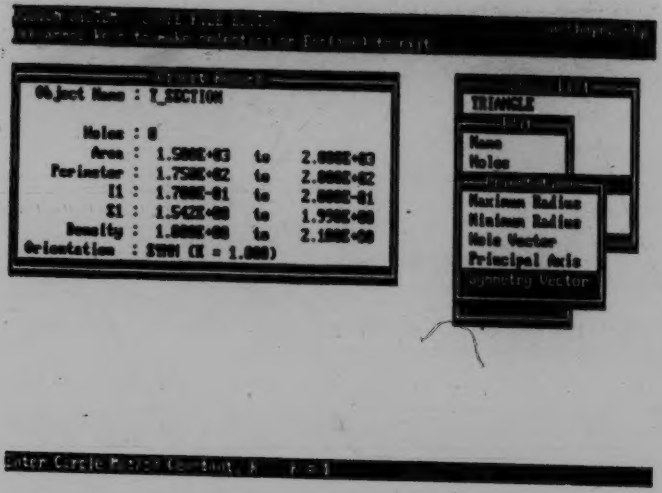
```
Main Menu                                         Press E, Q, C, F, M or Space

              E xposure
              C hange Camera
              F eatures
              M ean feature value calculation
              Q uit
              Space Bar : Take New Image

 Number of Pictures for calculating average FEATURE values : 0

Camera Number : 2                                 Exposure (ms): 100
```

Figure 4.3: The Local mode image acquisition phase.



```
Features                    Object Number : 2                 Camera 2

  Name       : T_SECTION
  X centroid :  3.102E+02mm
  Y centroid :  4.278E+01mm
  Area       :  1.721E+03mm^2
  Perimeter  :  1.912E+02mm
  Holes      :          0
  I1         :  1.746E-01
  S1         :  1.690E+00
  Density    :  1.916E+00
  Ref Axis   : -1.317E+02
```

Figure 4.4: The Local mode image analysis or 'features' phase.

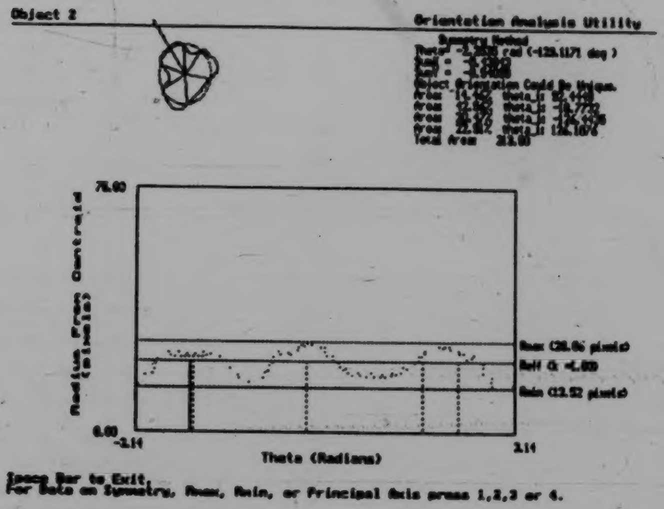


Object Name : T_SECTION

Holes : 0
Area : 1.500E+03 to 2.000E+03
Perimeter : 1.750E+02 to 2.000E+02
I1 : 1.700E-01 to 2.000E-01
S1 : 1.542E+00 to 1.990E+00
Density : 1.000E+00 to 2.100E+00
Orientation : SYMM (K = 1.000)

TRIANGLE
Name
Holes
Maximum Radius
Minimum Radius
Hole Vector
Principal Axis
Symmetry Vector

Figure 4.5: A typical Shape Edit session screen.



Object 2
Orientation Analysis Utility
Symmetry Method
70.00
0.00
-3.14
3.14
Theta (Radians)
Radius From Centroid (pixels)
Rmax (28.06 pixels)
Rref (K =1.000)
Rmin (13.52 pixels)

Theta: [illegible] rad (-128.1171 deg)
[illegible]
Object Orientation Could Be Unique.
Area [illegible] theta_i: [illegible]
Area [illegible] theta_i: [illegible]
Area [illegible] theta_i: [illegible]
Area [illegible] theta_i: 126.1076
Total Area [illegible]

Space Bar to Exit.
For Data on Symmetry, Rmax, Rmin, or Principal Axis press 1,2,3 or 4.

Figure 4.6: A typical ORIENT orientation method analysis phase.

```
                Perimeter,
                I1,
                S1,
                Density : Range;
                case Method: ObjectName of
                'SYMM': (K:REAL);
            end;
```

The quantities `I1`, `S1`, `Density` represent the first moment invariant, compactness and elliptical area density descriptors respectively (See appendix B. Note that `Method` can be one either `HOLE`, `PRAX`, `MAX`, `MIN` or `SYMM`, meaning the Hole, principal axis, maximum radius, minimum radius and circle method (or symmetry method) respectively, used to determine object orientation.

The ORIENT Orientation Utility

In VISYS, object orientation is determined after object identification and the orientation method incorporated into the object's record. An orientation method is chosen based on results produced by the ORIENT orientation utility program. Within ORIENT the user may compare the perimeter and inertia based orientation methods and choose the most suitable strategy for a given object. See figure 4.6. This choice is then inserted into the object's record using the shape editor.

The Shape Editor

The menu driven editor allows the user to create, update, and delete object records from the object library. An object may be selected, a range value or orientation method altered and the results saved. The effectiveness of a change may then be immediately observed by reentering the local vision system and reexecuting the picture/features cycle.

The Remote Mode

The third and most important category of operations is the Remote Mode of operation. In the remote mode a host computer, linked to the vision system by serial line, may request one of three options.

Acknowledge : ('0') enables the host to check for the presence of the vision system. The system responds with the string: "Ok".

Picture : ('1') instructs the vision system to capture an image and must be immediately followed by the camera number and exposure time in milliseconds. The system responds with a string composed of the object number (as seen in the field of view), identification string, x and y centroid locations (relative to the camera) and the orientation angle in degrees.

Disengage : ('-1') shuts down the remote mode of the vision system. The system responds by returning to the main vision system menu.

4.2.3 The Vision Processor: VISYS

The vision software module is a modified version of Warkentin's [49] Turbo PASCAL/Assembler binary vision software. Though the base image processing is largely unchanged, modifications were made to implement this software for use in a work cell environment. The original system, designed to perform in isolation and provide object identification, location and, to some degree, orientation (only within +/-180 degrees), was modified to provide remote service, accurate orientation for assembly, and a useful object editor. The vision software now performs the following duties:

Teaching with an interactive teach utility for new object description.

Debugging with a menu driven editor for existing object descriptions.

Operation with a simple remote vision instruction set returning location and orientation in camera coordinates.

The image processing modules can be divided into three operations

- image preparation and connectivity
- Object Identification
- Object Placement and Orientation

Image Preparation and Connectivity

Warkentin's Assembler routines perform image acquisition, storing the image as an array of 8192 bytes. Image noise is removed by applying *shrinking* and *growing* operators to the image. Growing a lone 'on' pixel results in turning all neighbouring pixels 'on' in the final image. If a pixel is to remain 'on' in the final image, all neighbouring pixels must be 'on' during the shrinking process. An *enhancement* stage rectifies peculiarities of the optic RAM and transforms the image into a 512 × 128 image. This image is converted into a Run Length Code array, where any continuous line of 'on' pixels is stored as an array element composed of a record containing the left and right terminal pixel positions. Connectivity analysis, or the derivation of *blobs* from the Run Length Code array, is performed using a published algorithm [44].The result is a set of associated records compactly describing blob dimensions.

A list of points on the perimeter is generated by collecting the terminal points of associated run length records in a clockwise fashion. The result is a list of x and *y* values describing the object perimeter. Further, hole perimeters are appended to the object perimeter list, separated by a preset number of null perimeter values. The number of perimeters contained in the list less the object perimeter is the number of holes in the object.

Object Identification

Object identification requires the use of parameters unaffected by object location or orientation. Typical values employed in binary vision are area, principal moments of inertia, density, perimeter, holes, and moment invariants such as polar moment of inertia. The connectivity records and perimeter lists are used to determine these properties using common algorithms found in [47] and [45]. A summary of these algorithms can be found in Appendix B.

During the teach phase, these values are stored in an object library as a range two standard deviations from mean trial values. During the operational phase the computed values are compared to those in the object library. In VISYS an object is identified when the computed values for moment invariants, density, area, and perimeter are within the legal bounds of an object in the object library.

Object Placement and Orientation

Once identified, the object centroid becomes the position of the object in the camera view area. Orientation is more difficult. Though a number of authors propose principal moments of inertia as referents for orientation, principal moments are only accurate to ±180 deg and, therefore, are only useful for symmetrical objects of high aspect ratio. Pugh [45] recommends the use of one of the following:

Holes where vectors are drawn between the object and hole centroids. The set of which may provide a unique orientation vector.

Radius where maximum or minimum perimeter radii may provide an orientation vector.

Circle Method where a circle of predetermined radius applied to the perimeter of the object will generate a set of intersection points. The direction of each may be described by a unit vector. The set of these vectors may provide a unique orientation vector.

VISYS uses all of these properties to determine object orientation. The preferred method for an individual object is determined by the user during the teach phase with the ORIENT utility package. This package allows the user to preview any method applied to any object. The chosen method is stored in the object library. Once the object has been identified, VISYS consults the library to determine the correct orientation method.

The Hole Method

The VISYS hole method sums vectors describing the hole centroid relative to the object centroid or

$$\vec{v}_{hole_i} = \bar{x}_{hole_i}\mathbf{i} + \bar{y}_{hole_i}\mathbf{j}$$

$$\vec{v}_{hole} = \sum_{i=1}^{n} \vec{v}_{hole_i} \qquad \text{where } n \text{ is the total number of holes.}$$

The vector sum of an asymmetrically distributed set of holes will produce a unique, repeatable vector. If the holes are symmetrically distributed the hole with the largest first moment of area relative to the object centroid is chosen as an orientation reference.

The Radius Method

The Radius method allows the user to select a single vector from the object centroid to a point on the perimeter. The direction of this vector, the radius, becomes a measure of orientation for objects with a uniquely large or small radius. A figure detailing both of these methods appears in figure 4.7.

The Circle Method

The effectiveness of the Circle Method depends on the quality of information stored in the perimeter. In VISYS a perimeter is essentially the endpoints of the run length code array for that object. Unfortunately, this is a poor match for the Circle Method as described in Pugh [45] which is sensitive to small variations of the perimeter near the circle boundary. Pugh points out that the diameter of the circle should be chosen such that:

- small changes in circle diameter do not effect the location of intersection points.
- small changes in circle diameter do not create or destroy intersection points.

As a possible corollary, the perimeter algorithm should produce:

- unique intersection point locations.
- repeatable intersection point locations.

Since these requirements depend on a smooth perimeter, a rough perimeter is often plagued with spurious and inconsistent intersection points near the circle boundary. Though smoothing algorithms could be employed to reduce these variations, they would also lengthen an already prolonged vision processing cycle [45]. Figure 4.8

shows a simplified image acquisition and perimeter generation, demonstrative typical perimeter roughness.

Briefly, the modified circle method employed in VISYS generates unit vectors in the direction of object areas *outside* the applied circle, providing a method of evaluating the quality of intersection pairs. By computing the angle between intersection points on the circle boundary, the effect of random variations of the perimeter near the circle boundary can be monitored and, if necessary, reduced by weighting each contributing vector by an angle quantity.

As a first step the Circle Method is made invariant with object size by making the circle radius, R_{eff}, a function of the object area:

$$R_{eff} = \sqrt{\frac{kA_{obj}}{\pi}} \tag{4.1}$$

... where k is a user defined constant stored in the object library. As the routine marches around the perimeter, points i and $i+1$, of coordinates x_i, y_i and x_{i+1}, y_{i+1} (relative to the centroid) respectively, are used to generate a mean radius from the centroid:

$$R_i = \sqrt{x_i^2 + y_i^2} \tag{4.2}$$

$$R_{i+1} = \sqrt{x_{i+1}^2 + y_{i+1}^2} \tag{4.3}$$

$$\overline{R}_i = \frac{R_i + R_{i+1}}{2} \tag{4.4}$$

If the mean, $\overline{R}_i$, becomes larger than R_{eff}, the area *slice* bounded by R_{eff} and $\overline{R}_i$ of width $d\theta_i$ can be computed. Note that if the points straddle R_{eff}, a linear interpolation is used to generate the appropriate intersection point.

Using the dot product of the two radii to find $\Delta\theta$ ...

$$\cos\Delta\theta_i = \frac{x_i x_{i+1} + y_i y_{i+1}}{R_i R_{i+1}} \tag{4.5}$$

$$\Delta\theta_i \approx \sin\Delta\theta_i = \sqrt{1 - \cos^2\Delta\theta_i} \quad \text{for small angles,} \tag{4.6}$$

While $\overline{R}_i$ is larger than R_{eff}, the i^{th} angle slices are summed to form the j^{th} angle *segment*, τ_j, composed of m total angle slices.

$$\tau_j = \sum_{i=1}^{m} \Delta\theta_i \tag{4.7}$$

Simultaneously a unit vector in the mean direction of the i^{th} slice, $\overline{\theta}_i$, is added to a segment vector sum.

$$\overline{\theta}_i = \arctan\left(\frac{y_i + y_{i+1}}{x_i + x_{i+1}}\right) \tag{4.8}$$

$$v_{xj} = \sum_{i=1}^{m} \cos\overline{\theta}_i \tag{4.9}$$

$$v_{yj} = \sum_{i=1}^{m} \sin\overline{\theta}_i \tag{4.10}$$

A segment is complete when $\overline{R}_i$ falls below R_{eff}. This procedure is performed over the entire length of the perimeter.

The products of the n normalized segment vector components, v_{xj} and v_{yj} and segment angles, τ_j are then summed. A lower boundary placed on the segment size can be introduced to reduce excessive 'noise' contributed by small fluctuations of the perimeter near the circle boundary.

$$\left.\begin{array}{l} V_x = \sum_{j=1}^{n} \frac{v_{xj}}{|v_j|}\tau_j \\ V_y = \sum_{j=1}^{n} \frac{v_{yj}}{|v_j|}\tau_j \end{array}\right\} \quad \text{if } \tau_j > \text{some lower boundary} \tag{4.11}$$

$$\vec{V} = V_x\mathbf{i} + V_y\mathbf{j} \tag{4.12}$$

The vector sum, $\vec{V}$, becomes the orientation vector. Since this method can be used to determine object symmetry, it is sometimes referred to here as the 'symmetry' method. See figure 4.9.

Though this algorithm seems overly complicated, it allows for experimentation with different segment vector weighting schemes. For example: One trial method weighted segment vectors by the segment area outside the circle. The slice area, A_i defined as:

$$A_i = \frac{\Delta\theta_i}{2}(\overline{R_i}^2 - R_{eff}^2) \qquad (4.13)$$

and the segment area, S_j, defined as the sum of these area slices or :

$$S_j = \sum_{i=1}^{m} A_i \qquad (4.14)$$

The orientation vector is then weighted by the segment areas:

$$\left.\begin{aligned} V_x &= \sum_{j=1}^{n} \frac{v_{xj}}{|v_j|} S_j \\ V_y &= \sum_{j=1}^{n} \frac{v_{yj}}{|v_j|} S_j \end{aligned}\right\} \text{ if } S_j > \text{some lower boundary} \qquad (4.15)$$

and substituted into equation 4.12.

Inconsistent perimeter quality made this area weighting scheme less reliable than the angle weighted method described above.

In summary, the binary vision system may operate in two possible processing modes: Local and Remote. In both modes, standard moment invariant methods are used to identify objects stored in an object library. Once identified, the orientation method, chosen by the user during the teach phase, is applied to the object and an orientation generated.

In Remote mode, the vision system may be run as a remote device to an external computer, providing x, y, and θ locations of objects in the camera's field of view. Local mode may be used to both inspect picture quality and teach the vision system new object properties through the shape editor module.

Though important in a real world application, the hardware is ultimately only a collection of input/output devices in this intelligent robotic work cell prototype. The strengths and weaknesses of this work cell, while dependent on the hardware, lie predominantly in the Natural Language software. The following chapter will review, in detail, the design and implementation of the Natural Language interface to the University of Alberta's work cell.

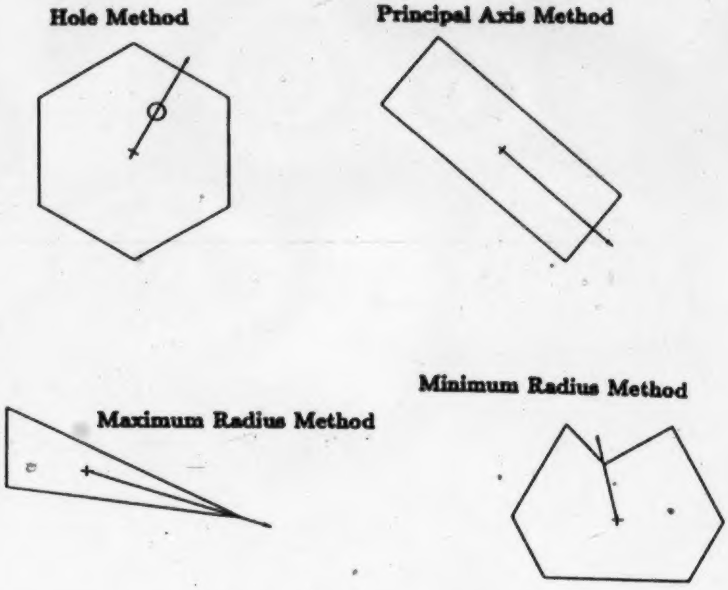
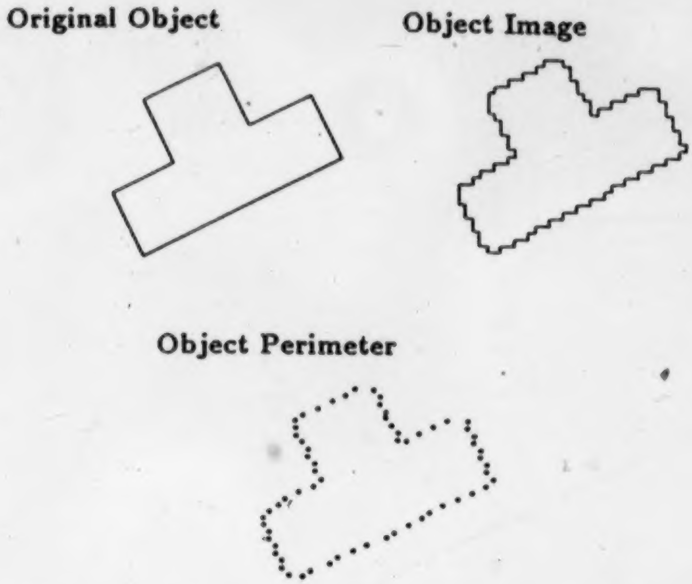

Hole Method
Principal Axis Method
Maximum Radius Method
Minimum Radius Method

Figure 4.7: The Hole, Radii and Principal Axis orientation methods used in the VISYS vision system.



Original Object

Object Image

Object Perimeter

Figure 4.8: The evolution of a typical perimeter in a VISYS image.

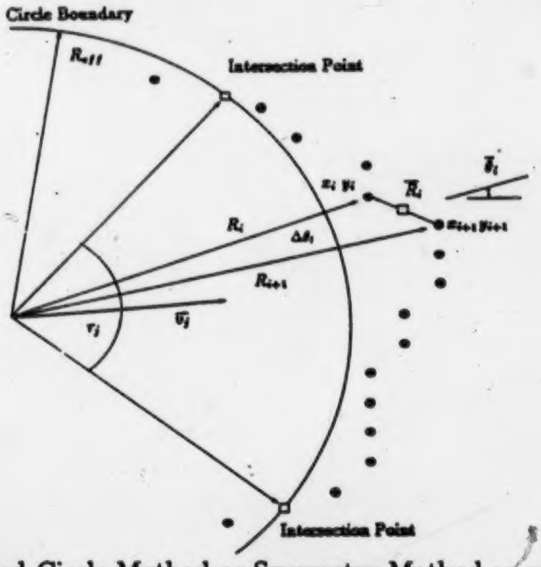

Circle Boundary
R_{eff}
Intersection Point
$x_i\ y_i$
$\bar{R}_i$
$\bar{\theta}_i$
R_i
$\Delta\theta_i$
$x_{i+1}\ y_{i+1}$
R_{i+1}
γ_j
$\bar{\psi}_j$
Intersection Point

Figure 4.9: The Modified Circle Method or Symmetry Method as employed by VISYS.

Chapter 5

The Natural Language Interface

This chapter will discuss, in detail, the design and implementation of the natural language interface: TSKMSTR. TSKMSTR investigates the minimum requirements of natural language processing for a robotic work cell. From the outset certain design requirements have been placed on the system. In brief, the design objectives for this first stage of work cell development are the following:

- to develop a syntactically based NLI prototype.
- to develop a recursive teach environment.
- to develop a simple Assembly world model.
- to incorporate a simple binary vision system.

5.1 TSKMSTR Operation

Before entering into a detailed discussion of this interface software, a series of sample dialogues are presented to give the reader a flavour of the TSKMSTR system.

The Environment

TSKMSTR, amounting to approximately 300kB of object code and operating on an 80386 PC with 640kB memory, divides the dialogue of the work cell between the user's natural language commands and system messages, each appearing in a window on the work cell monitor.

The 'Natural Language Command' window accepts both system 'keywords' and English command strings. The system recognizes the following seven keywords:

world invokes a brief textual list of world model objects, features and affixments.

log displays the system log containing a command history, command template data, and a trace of expanded predicates.

list displays a user selected file.

save saves all tasks to a user selected file.

consult loads tasks from a user selected file.

end terminates a task description in teach mode.

exit exits the NLI and shuts down the vision system.

The NLI window also displays reports generated during command execution. For example a `find` call prints an XYZRPY location to the screen. Similarly symbolic task equations, solved for T_6, appear during robot manoeuvres.

The System Messages window informs the user of errors, omissions, and basic system operations. Typical Message window outputs are: lexical or syntactic errors, vision and robot I/O, world model affixments, out of reach and out of range errors. The system also asks the user for clarification and confirmation of frames in the System Message window.

Upon input of a command string, acknowledged as syntactically correct in the Message window, one of three system responses are possible.

Execution : successful match with an existing library command acknowledged by an 'Ok' symbol.

Failure : either the command failed during execution or an element of the command is beyond the semantic scope of the NLI and not supported by the current work cell.

Teach : the string does not conform to an existing command in the library. Resulting in the prompt: "Do you wish to enter Teach Mode?"

Command Execution

Once the command string is parsed, the command is reduced to a *command template*, matched against the head of a subroutine in the command library, and executed in a command interpreter. The execution ultimately calls primitive work cell processes. The motion work cell primitives provide symbolic task equations reporting the progress of command execution.

Command Failure

Commands fail for three possible reasons: illegal syntax, undefined semantics, or execution failure. Illegal syntax simply implies a sentence that does not conform to the imperative grammar. The semantics of many syntactic constructs were deliberately left unaddressed in this prototype. A typical example of unaddressed semantics are descriptive adjectives, excluded since many adjectives describe qualities not measurable by the vision system or world model. Execution failure will occur if:

- the correctly specified coordinate frame cannot be found.
- the target frame is out of reach.
- a target frame is within reach, but out of the robot's joint range.

- the vision system has crashed.
- communications have failed.

Teach Invocation

If a command is legal but there is no matching routine, the user may the enter the teach phase. All teach dialogue occurs in a Teach Interface Window. For reference the unknown command is retained for reference in the middle of the screen. TSKMSTR subsequently forwards a series of 'yes/no' questions regarding the generality of the command. A 'yes' response to these questions flags frames as variables in ensuing user defined routines. Once flagged as a variable, all future references to that variable become parameters in the new assembly task.

The user then provides English commands describing the original unknown command. Should the user employ an unknown command during the explanation, the teach mode is reinvoked about the most recent unknown command. Another Teach Interface window is created and query session started.

Upon completion of an explanation, the user types **end** and the last Teach Interface Window is recovered. This cycle continues until no windows remain. The task definitions may then be saved to disk as PROLOG predicates through the **save** keyword.

Figure 5.1 presents a general flow diagram describing TSKMSTR operations.

5.1.1 Example Dialogues

The following example dialogues effectively illustrate TSKMSTR's operation and ease of use. Two forms of dialogue are possible: Executable and Teach. The

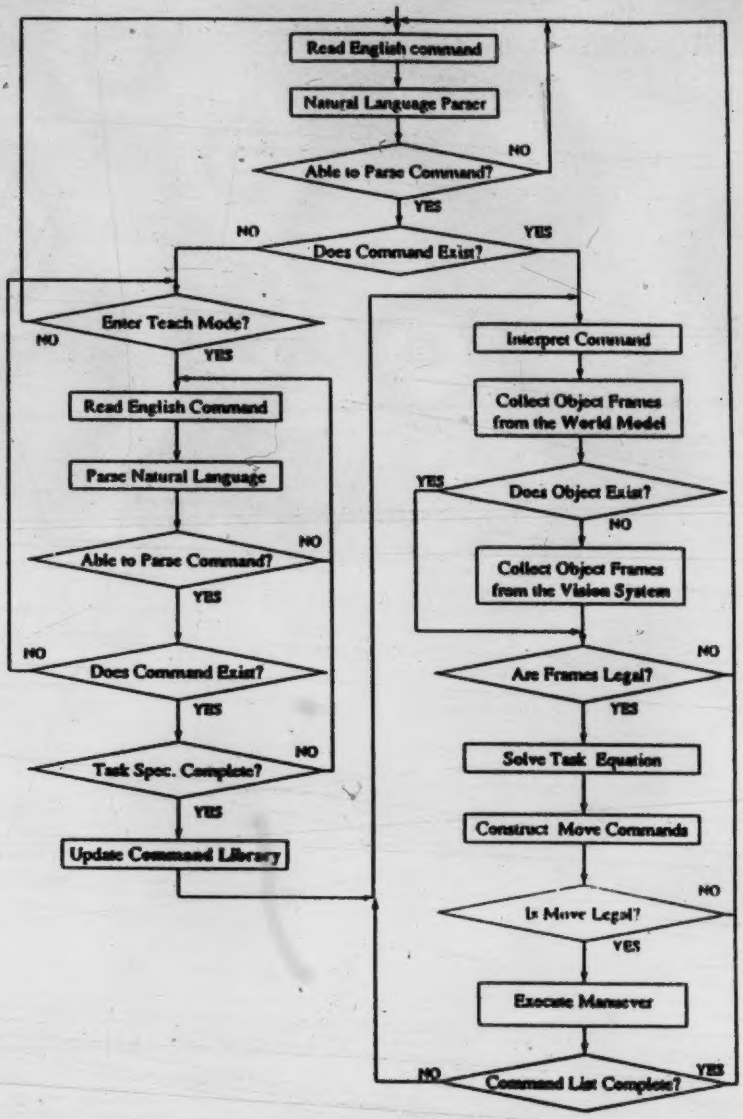

Read English command
Natural Language Parser
Able to Parse Command?
NO
YES
Does Command Exist?
NO
YES
Enter Teach Mode?
NO
YES
Read English Command
Parse Natural Language
Able to Parse Command?
NO
YES
Does Command Exist?
NO
YES
Task Spec. Complete?
NO
YES
Update Command Library
Interpret Command
Collect Object Frames from the World Model
Does Object Exist?
YES
NO
Collect Object Frames from the Vision System
Are Frames Legal?
NO
YES
Solve Task Equation
Construct Move Commands
Is Move Legal?
NO
YES
Execute Maneuver
Command List Complete?
NO
YES

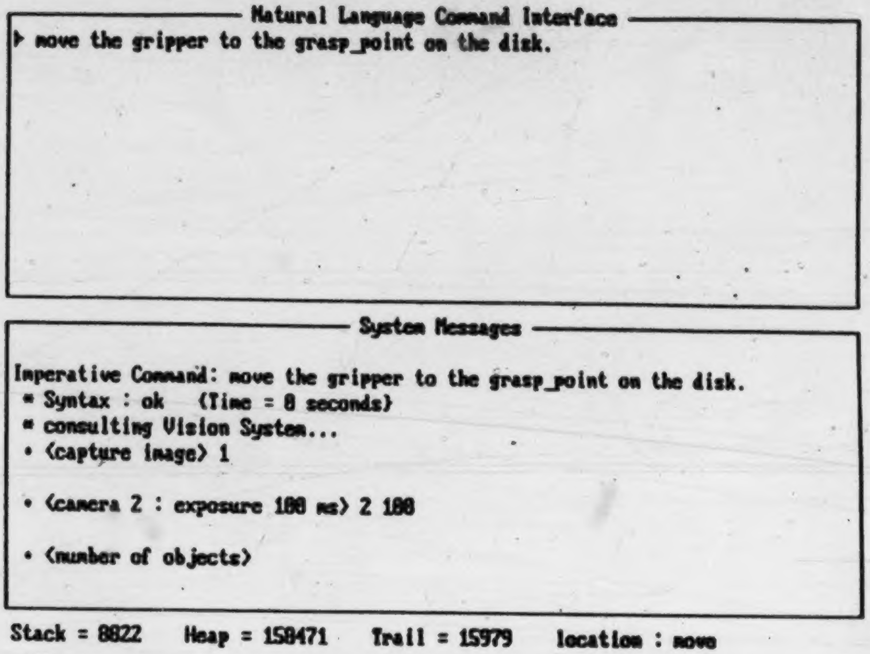
Figure 5.1: A flow diagram detailing the global operation of the TSKMSTR system.

executable dialogue shows the on–line behaviour of the TSKMSTR system, executing each command from the command line. The Teach mode is an off–line environment, allowing the user to define new commands in English without actually executing robot commands. The capability of the TSKMSTR system, though not as wide as that found in [17], is functionally comparable to the AL system with the benefit of an English syntax.

An Executable Dialogue

A typical command executed by TSKMSTR appears in figure 5.2. The command is parsed, converted to a command template, interpreted and reviewed by the kinematic inference engine. In this example, the disk cannot be found in the world model. The vision system is subsequently invoked and responds after approximately 10 seconds with the x, y and θ data string appearing in the Message Window of figure 5.3. This vision data supplies the world model with the necessary locational information, permitting execution to continue. The task equation describing the manoeuvre is built by the kinematic inference engine and presented beneath the command. Though the `i` denotes an identity matrix, the kinematic engine does not execute matrix inversion or multiplication processes on identities, avoiding unnecessary matrix multiplications. The computed robot joint angles appear in the Message Window.

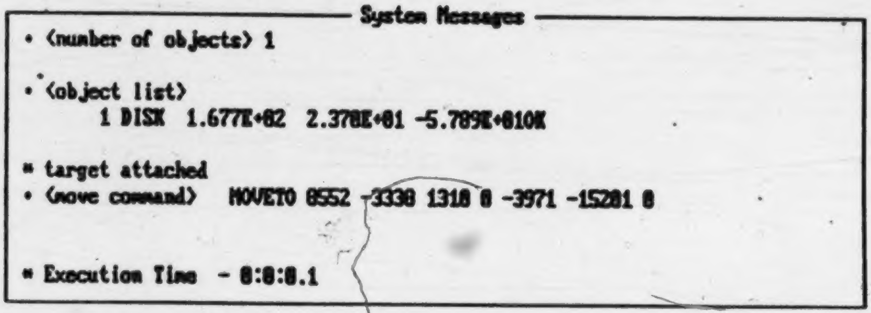
Once the gripper has been closed on the disk's grasp_point, the user `affixes` the gripper to the `grasp_point` on the disk, allowing the robot to move any portion of the disk. See figure 5.4. Any frame of the disk may be moved relative to its current position or to a new point in space. In figure 5.5 one such point has been `assigned` to a frame label by the user . Rotations may also be applied to the tool in the gripper. Figure 5.6 shows the execution of a `roll` manoeuvre. Note the adverb *anticlockwise*.



```
———————————— Natural Language Command Interface ————————————
▸ move the gripper to the grasp_point on the disk.
```

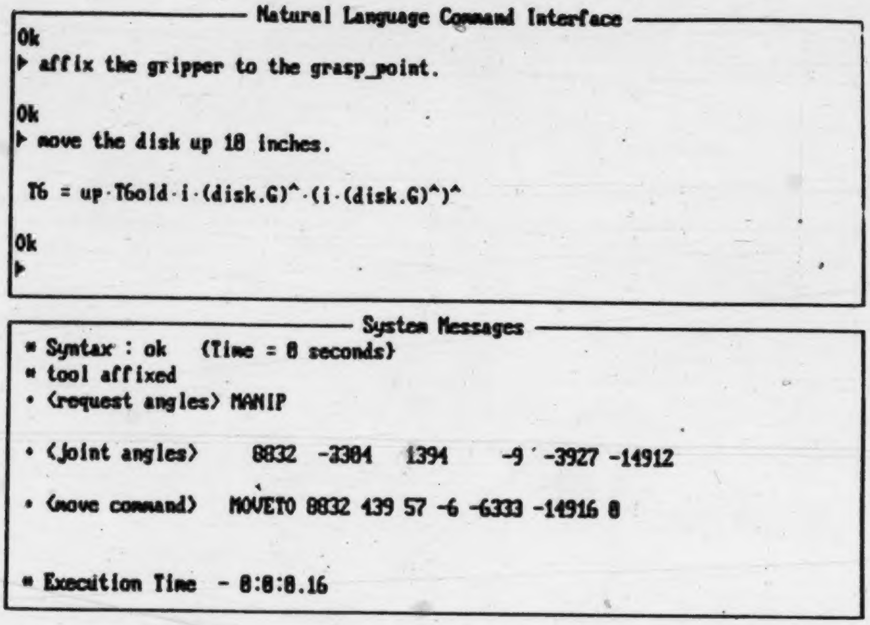
```
———————————————— System Messages ————————————————
Imperative Command: move the gripper to the grasp_point on the disk.
 * Syntax : ok   (Time = 0 seconds)
 * consulting Vision System...
 • <capture image> 1

 • <camera 2 : exposure 100 ms> 2 100

 • <number of objects>

Stack = 0022    Heap = 150471    Trail = 15979    location : move
```

Figure 5.2: Typical user command. Note that the vision system has been invoked.



```
———————————————— System Messages ————————————————
 • <number of objects> 1

 • <object list>
       1 DISK  1.677E+02  2.370E+01 -5.709E+010K

 * target attached
 • <move command>   MOVETO 6552 -3330 1310 0 -3971 -15201 0


 * Execution Time  - 0:0:0.1
```

Figure 5.3: Vision data, returned in the Messages window, is used in the execution of the user's command



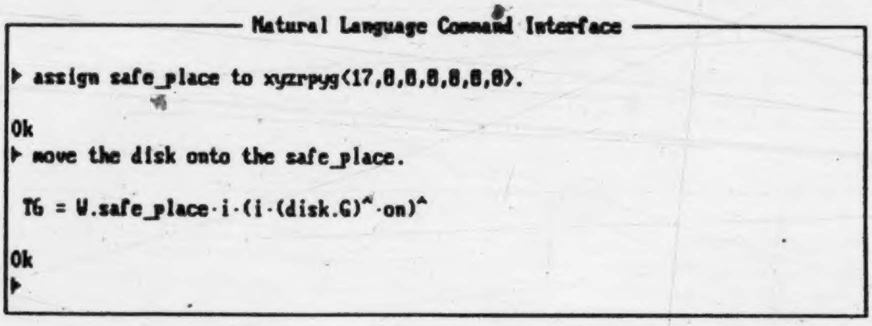
Natural Language Command Interface

```
Ok
▸ affix the gripper to the grasp_point.

Ok
▸ move the disk up 10 inches.

 T6 = up·T6old·i·(disk.G)^·(i·(disk.G)^)^

Ok
▸
```

System Messages

```
 * Syntax : ok   (Time = 0 seconds)
 * tool affixed
 • <request angles> MANIP

 • <joint angles>     8832  -3384   1394      -9  -3927 -14912

 • <move command>   MOVETO 8832 439 57 -6 -6333 -14916 0


 * Execution Time  - 0:0:0.16
```

Figure 5.4: With the gripper affixed to the disk, the disk may be moved relative to it's current position.

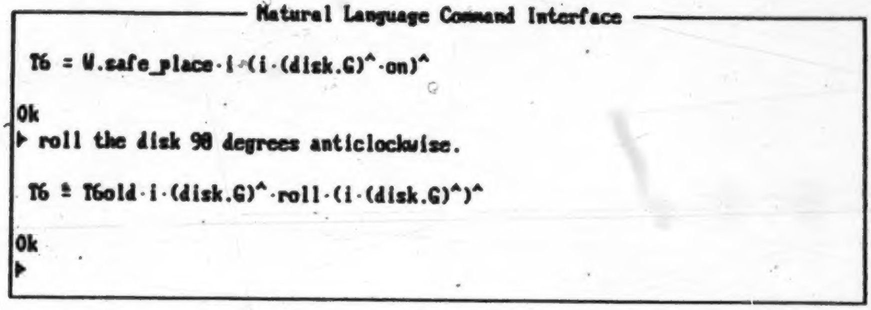


Natural Language Command Interface

```
▸ assign safe_place to xyzrpyg<17,0,0,0,0,0,0>.

Ok
▸ move the disk onto the safe_place.

 T6 = W.safe_place·i·(i·(disk.G)^·on)^

Ok
▸
```

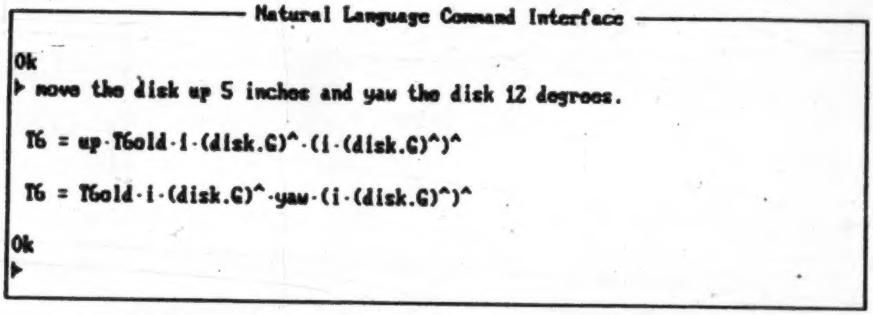
Figure 5.5: New labels may be made in the work cell through the **assign** task primitive. Objects may then be attached to these sites.

The grammar allows the use of compound sentences as in figure 5.7. Note that the order of execution agrees with the command structure. This performance is complicated by increasingly complex compound sentence structures as in figures 5.8 and 5.9 and requires the use of distribution functions. A final location report on selected work cell frames can be requested with the `find` task primitive as in figure 5.10.


Natural Language Command Interface

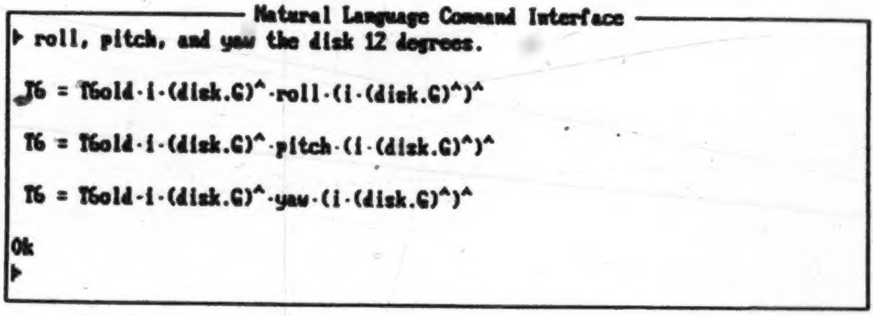
```
 T6 = W.safe_place·i·(i·(disk.G)^·on)^

Ok
▸ roll the disk 90 degrees anticlockwise.

 T6 = T6old·i·(disk.G)^·roll·(i·(disk.G)^)^

Ok
▸
```

Figure 5.6: An example of the `roll` primitive with the adverb `anticlockwise`. The processor also accepts signed degree angles.


Natural Language Command Interface

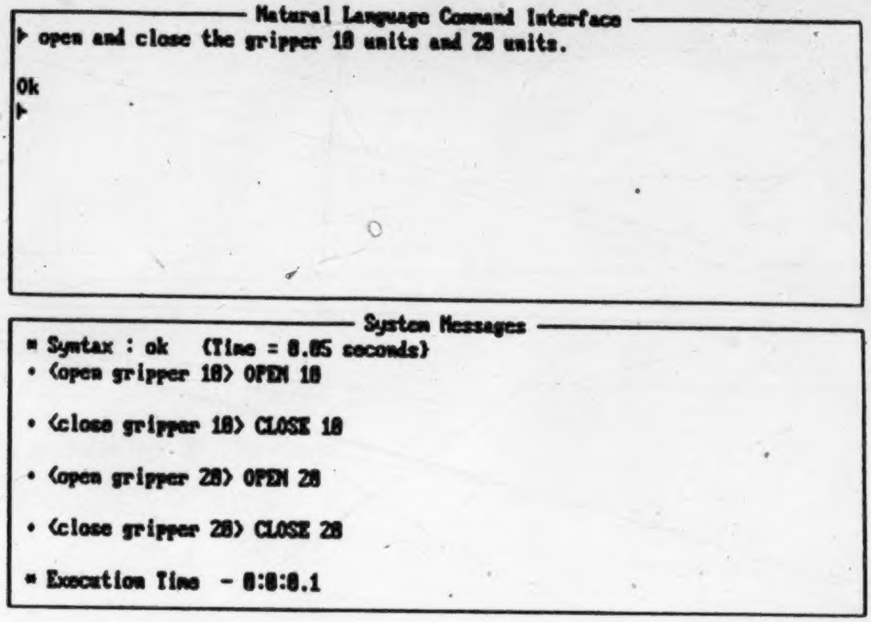
```
Ok
▸ move the disk up 5 inches and yaw the disk 12 degrees.

 T6 = up·T6old·i·(disk.G)^·(i·(disk.G)^)^

 T6 = T6old·i·(disk.G)^·yaw·(i·(disk.G)^)^

Ok
▸
```

Figure 5.7: An example of a compound sentence. Note the execution order implied by the task equations reported in the Interface window.


Natural Language Command Interface

```
▸ roll, pitch, and yaw the disk 12 degrees.

 T6 = T6old·i·(disk.G)^·roll·(i·(disk.G)^)^

 T6 = T6old·i·(disk.G)^·pitch·(i·(disk.G)^)^

 T6 = T6old·i·(disk.G)^·yaw·(i·(disk.G)^)^

Ok
▸
```

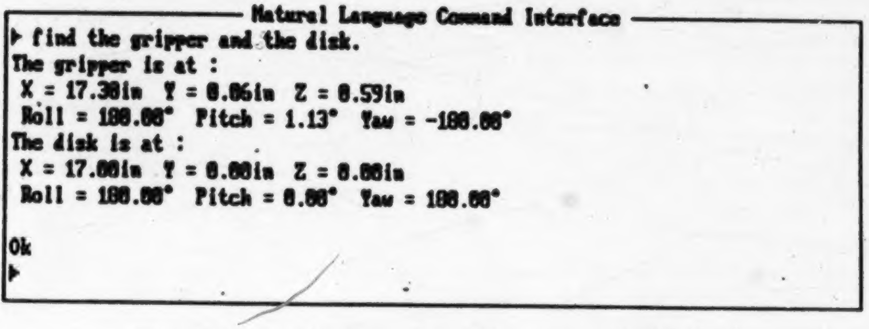
Figure 5.8: A more complex example of a compound command clause.


Natural Language Command Interface
System Messages

```
▸ open and close the gripper 10 units and 20 units.

Ok
▸
```

```
* Syntax : ok   (Time = 0.05 seconds)
• <open gripper 10> OPEN 10

• <close gripper 10> CLOSE 10

• <open gripper 20> OPEN 20

• <close gripper 20> CLOSE 20

* Execution Time  - 0:0:0.1
```

Figure 5.9: A worst case example of compound clauses. Once again, note the execution order.

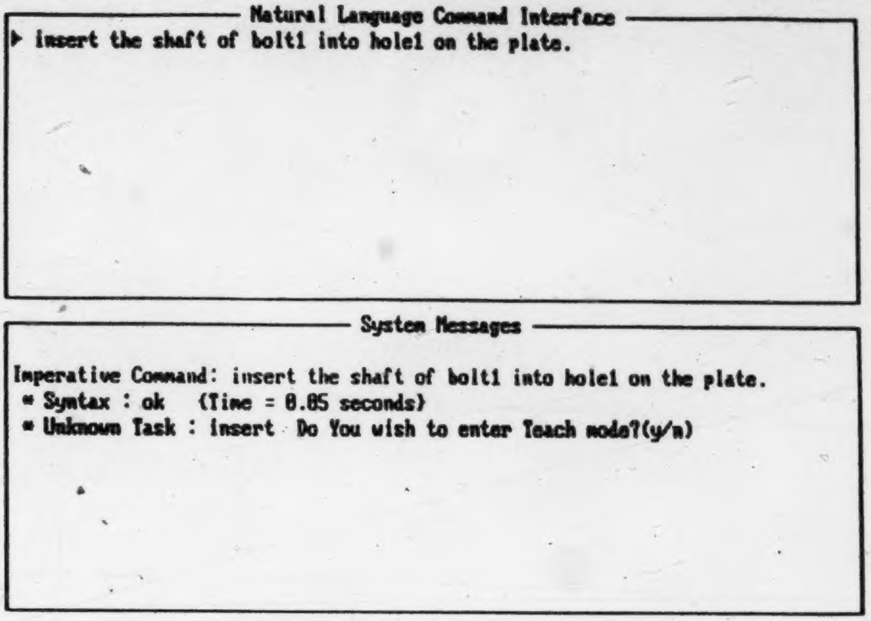

Natural Language Command Interface

```
▸ find the gripper and the disk.
The gripper is at :
 X = 17.30in  Y = 0.06in  Z = 0.59in
 Roll = 180.00°  Pitch = 1.13°  Yaw = -180.00°
The disk is at :
 X = 17.00in  Y = 0.00in  Z = 0.00in
 Roll = 180.00°  Pitch = 0.00°  Yaw = 180.00°

Ok
▸
```

Figure 5.10: An example of the `find` primitive.

A Teach Dialogue

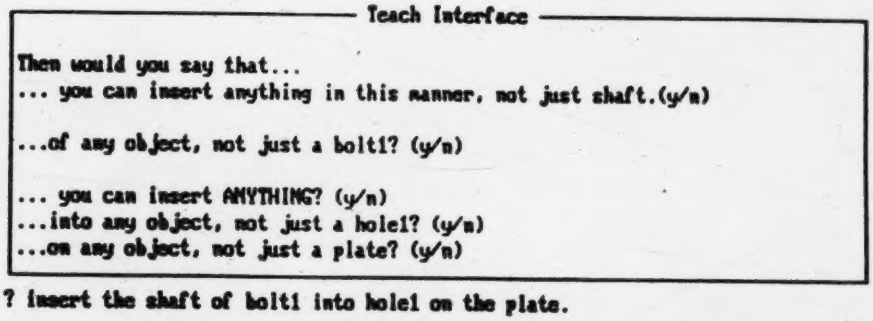
If an unknown command is issued, the teach mode is invoked as in figure 5.11. Through the query dialogue in figure 5.12, the user flags variables in the command template, effectively generalizing the command for use in other task descriptions (in this case the command is to be fully generalized). If one of the commands defining the unknown command is also unknown, the teach mode is reinvoked. See figure 5.13. After responding to another query session, the subcommand is described by the English commands in figure 5.14. The 'end' keyword concludes the teaching session allowing the user to 'ascend' and complete the original teach session (figure 5.15).



```
Natural Language Command Interface
▶ insert the shaft of bolt1 into hole1 on the plate.
```

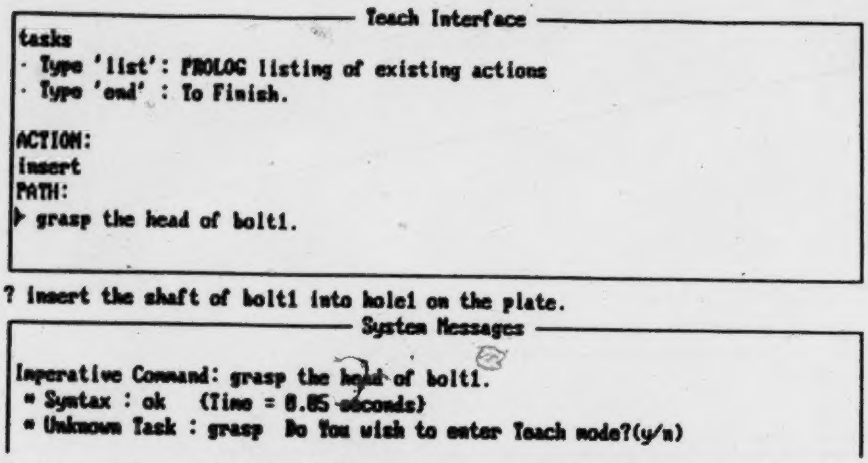
```
System Messages

Imperative Command: insert the shaft of bolt1 into hole1 on the plate.
 * Syntax : ok   (Time = 0.05 seconds)
 * Unknown Task : insert  Do You wish to enter Teach mode?(y/n)
```

Figure 5.11: The Teach mode is invoked.



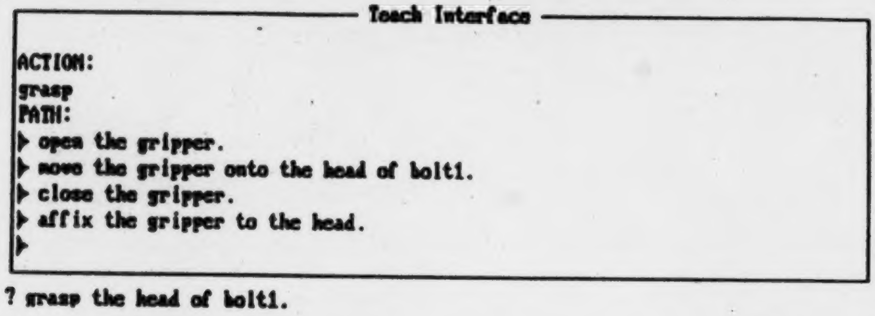
```
Teach Interface

Then would you say that...
... you can insert anything in this manner, not just shaft.(y/n)

...of any object, not just a bolt1? (y/n)

... you can insert ANYTHING? (y/n)
...into any object, not just a hole1? (y/n)
...on any object, not just a plate? (y/n)
? insert the shaft of bolt1 into hole1 on the plate.
```

Figure 5.12: The query engine asks the user to generalize the command.



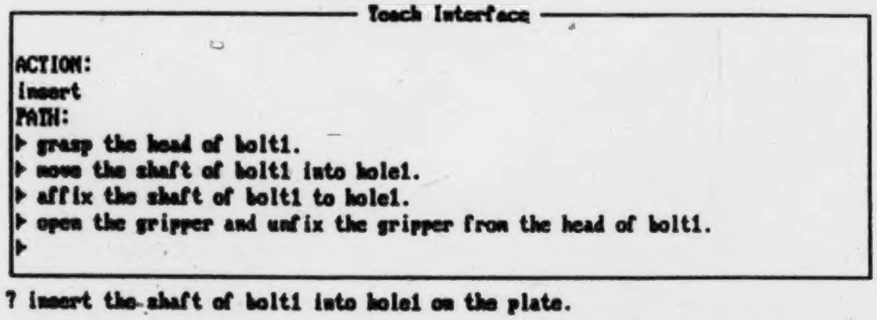
```
                              Teach Interface
tasks
· Type 'list': PROLOG listing of existing actions
· Type 'end' : To Finish.

ACTION:
insert
PATH:
▸ grasp the head of bolt1.

? insert the shaft of bolt1 into hole1 on the plate.
                              System Messages

Imperative Command: grasp the head of bolt1.
 * Syntax : ok   (Time = 0.05 seconds)
 * Unknown Task : grasp  Do You wish to enter Teach mode?(y/n)
```

Figure 5.13: During the definition of insert the Teach mode is reinvoked by a grasp command, not defined in this example.



```
                              Teach Interface

ACTION:
grasp
PATH:
▸ open the gripper.
▸ move the gripper onto the head of bolt1.
▸ close the gripper.
▸ affix the gripper to the head.
▸

? grasp the head of bolt1.
```

Figure 5.14: Grasp is defined. Note the inclusion of the affix command relating the gripper to the head of bolt1.



```
                              Teach Interface

ACTION:
insert
PATH:
▸ grasp the head of bolt1.
▸ move the shaft of bolt1 into hole1.
▸ affix the shaft of bolt1 to hole1.
▸ open the gripper and unfix the gripper from the head of bolt1.
▸

? insert the shaft of bolt1 into hole1 on the plate.
```

Figure 5.15: The teach mode 'ascends' back into the insert teach session.

5.2 TSKMSTR Design and Implementation Overview

Designed to be a prototype NLI, TSKMSTR uses a context free grammar and basic natural language processing techniques leaving optimization issues to later efforts. A successful parse is converted into a *command template* by the *semantic processor*. This command template is then passed to a *command interpreter* that executes the procedure.

TSKMSTR's task definitions, similar to those described by Bock [15], Selfridge and Vannoy [17], defines new commands in terms of existing task definitions. Unlike other robotic NLIs, however, a teach phase that generalizes tasks through variable arguments has been developed to maximize the usefulness of NLI routines. The teach phase is recursive, i.e any English command not recognized during the explanation of a new task will reinvoke the teach mode.

During the execution of object level *task primitives*, objects are found through the inspection of an *assembly world model*. This world model incorporates characteristics typical of object level robot languages such as the linked list representation and frame affixment schemes of AL [10] and SRL [14]. This prototype work cell relies on the conjunction of coordinate frames between manipulator tools and world targets to construct 'blocks world' assemblies. The world model is essentially 'hidden' from the user, who may concentrate on assembly instead of manipulator procedures.

If an object cannot be found in the world model, the binary vision system inspects the work area for new objects, providing new data to the world model spontaneously, without instruction by the user.

To generate robot level instructions, these object locations are placed into task equations, as outlined in Paul [22], and solved for T_6. The resultant joint angles determined through the inverse kinematic solution are then transmitted to the robot via the serial communication module.

In summary, TSKMSTR is composed of the following five modules

- The Scanner and Parser
- The Semantic Processor
- The Command Interpreter
- The World Model, Kinematic Engine and Task Primitives
- Math Utilities and Communication

For reasons that will be detailed later, the bulk of these routines have been written in Turbo PROLOG [40], while the math utilities have been written in 'C'. For further information on PROLOG, consult Bratko [39]. The following sections will describe the design and operation of each module in detail.

5.3 The Scanner and Parser

In this prototype system, a simple syntactic Context Free Grammar has been adopted to model the input language. Based largely on the grammar presented in Bock [15], the PROLOG source code for the parser was generated by an off–the–shelf parser generator utility [41].

5.3.1 The Grammar

Bock's CFG modelled the Imperative command subset of the English language. Though adequate for Bock's purposes, some modifications were necessary to permit specification of vectors and to alleviate the semantic processing requirements of prepositions. Furthermore a 'work order' approach to English language instructions combined with empirical results of early trials indicated that some segments of the grammar were rarely used and were either dropped from the grammar or left unsupported in the semantic processor.

The 'work order' approach to English instructions supposes that the user has English language list of procedures required to complete an Assembly, a philosophy similar to that used in the description of AUTOPASS [8]. These assembly instruction sheets are usually composed of imperative command statements, and seem to possess little or no 'literary' structures such as dependent clauses, infinitive, gerundial or participial phrases. Commands are often simple or compound predicates containing noun phrases within compound indirect and direct objects. Modifiers range from simple adverbs and adjectives to conjoined sets of prepositional phrases. Only the conditional form of dependent clause is supported by the semantic processor (though not currently supported by TSKMSTR's command interpreter as explained later).

5.3.2 Vectors

Bock's grammar lacked any method of stating explicit positions in the world. This gap was filled through the creation of a noun-like *vector* non-terminal. The vector is currently defined as either a one, two, three or seven element list of real numbers separated by commas. The seven element list allows either explicit robot joint positions or the six quantity location/orientation XYZRPY construct plus a gripper jaw width.

5.3.3 Prepositions

Though the bulk of the terminals in Bock's grammar are well defined symbols such as nouns, verbs, adverbs and adjectives, a failing of his grammar is the poor definition of prepositional phrases. An easily recognized characteristic of colloquial English is that prepositional phrases have highly ambiguous semantic values. For example the command,

Place the keys on the ring in my pocket.

is ambiguous, having two possible meanings:

- Place the keys onto the ring that is inside my pocket,
- Place the keys on the ring into my pocket.

Since it is perfectly possible that the world model may contain data supporting both interpretations, a semantic convention developed by Bennett [52] was incorporated into the grammar to clearly define the meaning of prepositional phrases.

Bennett isolates spatial prepositions from their contextual semantics and develops a method of describing prepositional meaning through the combinations of five cases. These cases are *locative, source, path, goal* and *extent* Our interest, however, lies in the definition of the first four.

Locative prepositions define a location. It is important to reiterate that Bennett's analysis is isolated from context, thus eliminating the natural assumption that a *locative* preposition is of necessity a location *in space.* For example:

He saw the car *in the showroom.*

is plainly a location in space while

He saw the car *in his dreams*

is not. This typifies the analytical style adopted by Bennett to characterize prepositional meaning. The following explanations will show through examples how Bennett's classifications are flexible across apparent semantic barriers.

Source prepositions define some starting point as in

Newton got the idea when an apple fell *from a tree.*

where *from* denotes a starting point in space and

Newton got the idea *from a falling apple.*

implies some event.

A path preposition implies some transitional state as in

She sent the package to Paris *via Amsterdam.*

Here *via* denotes passage through some intermediate point in space and in

She sent the package to Paris *via air mail.*

describes a mode of transport.

The goal preposition defines an end point as in

He followed his father *into the boat.*

where *into* implies a physical end point and

He followed his father *into politics.*

implies a career goal.

Bennett employed *componential analysis* to define the precise meaning of a given preposition. Combinations of *locative, source, path* and *goal* were used to form a semantic definition of a preposition. For example, the preposition *over* may be used in many ways such as:

1. He placed his hand *over the money.*
2. The vineyard is *over the hill.*

In sentence one, *over the money* is plainly a location in space, though above the location of *the money.* This sense of *over* is therefore some position (locative) related to another position by an implied hierarchy or formally:

locative superior

In the next sentence *over the hills* implies a position(locative) related by a path (path) through some other hierarchical position (locative) or:

locative path locative superior

The analysis amounts to reviewing several phrases of a given preposition and discovering common denominators in their semantics e.g. for *over* the result becomes:

locative superior

A group of thirty similarly analyzed prepositions are shown in tables 5.1 and 5.2.

The semantic values of *locative, source, path* and *goal* were appended to the grammar thus clarifying any prepositional ambiguity. Since these classifications are based on colloquial English semantics, the restriction imposed by this convention is relatively unnoticeable and greatly simplifies semantic post processing. For source prepositions only *from* is recognized, while path prepositions and goal prepositions are (*through, via*) and (*into, onto, to*) respectively. Locative prepositions were limited to (*in, on, at*).

Verbals

The Verbals (gerunds, participles and infinitives) proposed in Bock's grammar are rarely used colloquially and have been dropped from the grammar. The philosophy of the TSKMSTR approach has been to employ a 'work order' level of instruction, which tends to be a straight forward master/slave dialogue. The use of such 'literary'

Table 5.1: Bennett's [52] analysis of TSKMSTR's locative prepositions.

Preposition	Componential Analysis
at	locative
in	locative interior
on	locative surface

Table 5.2: Bennett's [52] analysis of TSKMSTR's non-locative prepositions.

Preposition	Componential Analysis
from	source
via	path
to	goal
onto	goal locative surface
into	goal locative interior

phrases seems unlikely in the work place.

Participles are adjectival verb forms that may possess objects. TSKMSTR takes a very narrow interpretation of adjectival forms and limits modifiers to simple adjectives and locative prepositional phrases, since these are the most common forms in a master/slave dialogue. It is worth noting that in Bock's grammar, the limited variety of participles (having been, having, being) is an implicit recognition of the narrow application these 'literary' structures have at this level of robotic instruction. Furthermore, the use of participles is often avoidable. For example:

Having opened the gripper, grasp the box.

is essentially identical to :

Open the gripper and grasp the box.

It seems that in a master/slave dialogue, the latter is a more likely construct. For these reasons participles were dropped from the grammar.

Since the teach system will query the user if a task is unknown, the use of an infinitive phrase such as:

To start the engine, rotate the keys to the on position

for instruction is redundant. In TSKMSTR the unknown instruction

Start the engine.

is sufficient to enter the teach phase. It is for this reason that infinitives were removed from the grammar.

Though gerundial phrases such as

Moving the gripper to the box, open the gripper.

are fairly common, the subordinating conjunction implies a parallel task execution, not currently available in the work cell hardware/software package.

The resultant grammar appears in appendix D.

5.3.4 The Parser Generator

The parser generator converts CFGs, described in a slightly modified Backus Naur Form notation, into an efficient PROLOG Definite Clause Grammar (DCG) parser [54], [55]. Briefly, the concept of a DCG stems from the realization that a context free rule can be converted into a logical statement: the *Horn Clause*. The Horn clause, also called a *regular* or *definite* clause is composed of a *head* and a *body*. For example

Mammoth → Large and Hairy and Tusks.

This clause may be interpreted as a logical statement in which the head, on the left side, is true if the body, on the right, is also true. These clauses may be interpreted procedurally, verifying each member in the body verifies the head.

Sentence → Subject and Verb and Object

Each member of the body is verified, from left to right, in a top down, depth first process. The Horn Clause is also the foundation of the *predicate* in the PROLOG programming language. A context free grammar rule may therefore be converted via the DCG formalism into a PROLOG predicate. An example is outlined by Mehdi [51].

Most DCG parsers employ a difference list method of parsing. The head of a given DCG rule contains input and output parameters. The successful verification of body predicates results in the *unification* or construction of head output parameters. Typically the input parameters are composed of a list of *tokens* or words, while the output parameters are PROLOG record structures called *compounds* describing the syntax of the string. The predicates in the body attempt to confirm that portions of the input token list represent portions of the legal language. For example the sentence

Bill likes Jane.

becomes the token list:

```
[noun("Bill"),verb("likes"),noun("Jane")]
```

and when inserted as the term `TOKEN_LIST` into the following parser:

```
sentence(TOKEN_LIST,D_LIST,syntax(SUBJ,VERB,OBJ)) :-
    subject(TOKEN_LIST,D_LIST1,SUBJ),
    verb(D_LIST1,D_LIST2,VERB),
    object(D_LIST2,D_LIST,OBJ).

subject([noun(WORD)|REST],REST,s(WORD)):- !.

verb([verb(WORD)|REST],REST,v(WORD)):- !.

object([noun(WORD)|REST],REST,o(WORD)):-!.
```

...is parsed as:

`syntax(s("Bill"),v("likes"),o("Jane"))`

Note that the `D_` prefix on the predicate parameters are difference lists and contain the remainder or `REST` of a tokenlist after a successful token match.

The parser generator produces a still more efficient form of DCG parser. Often production rules contain repetitive references to a single nonterminal. For example in the following grammar COM_VERB_EXP must be discovered twice if COM_VERB_PHR does not exist.

COM_VERB_PHR → COM_VERB_EXP CONN COM_VERB_PHR |
COM_VERB_EXP

To solve this problem, the parser generator converts this BNF representation into the following form of DCG PROLOG representation:

```
s_com_verb_phr(LL1,LL0,CVP) :-
     com_verb_exp(LL1,LL2,CVE),
     s_com_verb_phr1(LL1,LL0,CVE,CVP)

s_com_verb_phr1(LL1,LL0,CVE,cvp1(CVE,CONN,CVP)):-
     conn(LL1,LL2,CONN),
     s_com_verb_phr(LL2,LL0,CVP),!.

s_com_verb_phr1(LL,LL,CVE,cvp2(CVE)) :- !.
```

This form allows the derivation of **COM_VERB_EXP** (CVE) to be preserved should no further **COM_VERB_PHRs** (CVP)be found.

The DCG also allows the grammar to be sensitive to context through inspection of the token list for upcoming, perhaps significant, tokens. For production rules with consecutive terminal symbols, such as a set of numbers e.g.

VECTOR → real, real, real

The generated parser 'looks ahead' using an `expect` function that simply looks at the next values on the list.

```
vector([t(real(REAL),_)|LL1],LL0,dim3(REAL,REAL1,REAL2)):-
        expect(t(comma,_),LL1,LL2),
        expect(t(real(REAL1),_),LL2,LL3),
        expect(t(comma,_),LL3,LL4),
        expect(t(real(REAL2),_),LL4,LL0),!.
```

Note the structure of the token list. In TSKMSTR commands are conveyed to the system from the keyboard. Upon reception from the keyboard, an input string is passed to a *scanner*.The scanner dissects the sentence into individual tokens that are, in turn, identified with certain standard syntactic values contained in a *dictionary*. The dictionary, a PROLOG database, classifies words through a string matching process. The word classes are modelled by two PROLOG compound data structures:

`w(STRING,CLASS)` : where `STRING` is the word and `CLASS` is the word class where a word may have only one class e.g.

`w("gripper","noun")`

`dual(STRING,CLASS)` : where `STRING` is the word and `CLASS` is a word class and where a word may have multiple classes e.g.

Table 5.3: A comparison between TSKMSTR and traditional English word classes

English Classification	TSKMSTR	Example
transitive and intransitive verbs	verb	move, insert
adverbs and particles	adverb	completely, up, down
nouns	noun	box, inches
	vector	xyzrpyg, position
adjectives	adjective	blue
	real	5.2, −1.5
prepositions	source	from
	path	via, through
	goals	to, into, onto
	location	at, in, on
definite and indefinite articles	article	the, a
subordinating conjunctions	subconj	if, until
coordinating conjunctions	coordconj	and ','

```
dual("open","adjective")
dual("open","verb")
```

Table 5.3 shows the list of word classes.

During the scanning phase when a token is identified as having the `dual` class, the token is identified as a `word` of undetermined class. The classification of the `word` is defined by the parser. For some classes such as nouns, verbs, and adjectives `dual` tokens are potentially quite common. Each of these classes possess a parsing predicate capable of confirming the membership of a `word` within a `dual` of a particular class. For example the verb "open", initially identified by the scanner as class simply `word(open)`, is successfully parsed by the verb parse predicate if a compound `dual("open","verb")` can be found in the database.

Once matched, each word is inserted into a PROLOG compound structure, along with an integer indicating the word's location in the sentence.

These compounds are subsequently inserted into the token list. For example, the sentence *Completely affix the peg to the hole.* becomes the following token list after scanning.

```
[
t(adv(completely),0),t(verb(affix),11),
t(art(the),16),t(n(peg),20),
t(gl(to),24),t(art(the),27),t(noun(hole),32),t(period,34)
]
```

This list structure is then passed to the parser. An example of the parsed command appears in figure 5.16.

```
com_str2(
  command1(
    mod4(
      modgrp3(
        advexp2(
          adv1("completely")
        )
      )
    ),
    imppred3(
      cvp2(
        cve2(
          verb1("affix")
        )
      ),
      do2(
        np2(
          ng2(
            noun1("peg")
          )
        )
      )
    ),
    mod4(
      modgrp2(
        prep4(
          gl3(
            to("to",
              np2(
                ng2(
                  noun1("hole")
                )
              )
            )
          )
        )
      )
    )
  )
)
```

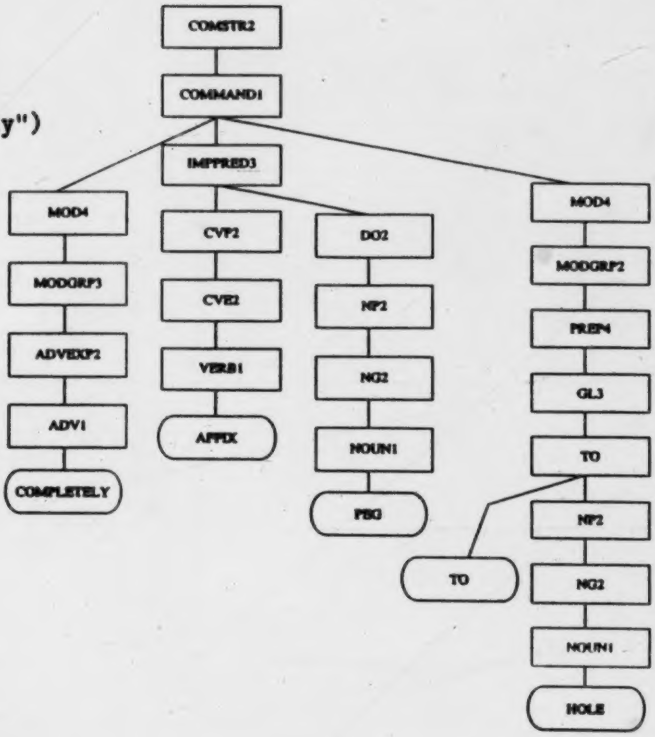

COMSTR2
COMMAND1
IMPPRED3
MOD4
CVP2
DO2
MOD4
MODGRP3
CVE2
NP2
MODGRP2
ADVEXP2
VERB1
NG2
PREP4
ADV1
AFFIX
NOUN1
GL3
COMPLETELY
PEG
TO
NP2
TO
NG2
NOUN1
HOLE

Figure 5.16: The parse structure for *Completely affix the peg to the hole.*

5.4 The Semantic Processor

Once the command string has been parsed, the structure of the sentence is formalized and the meaning derived in a semantic processor.

In general, the semantic processor closely parallels the execution of the parsing module and, in fact, could be incorporated directly into the parser. However, frequent changes to the grammar and subsequent regeneration of the parser require that, for this prototype NLI, the parser and semantic processor should remain separate. The processor's activities can be divided into three categories:

- the decomposition of the parse structure into key syntactic substructures
- the assertion of key substructure features into syntax registers
- the retraction and distribution of the syntactic registers into command templates.

The semantic processor employs a set of environment *register* stacks during the construction of a command's meaning. At any time in the semantic processing procedure these environment registers contain a record of useful syntactic structures. Essentially the semantic processor peels away the 'outer' structures to reveal key syntactic substructures. As each substructure is isolated, features characteristic of the substructure are 'pushed' or 'asserted' onto an environment register stack to be 'popped' or 'retracted' for collection and correlation at a later stage.

In a semantic processor predicate, an input syntactic structure is decomposed into increasingly fundamental structures by processors in the predicate body. For example:

```
comverbphr(cvp1(COM_VERB_EXP,CONN,COM_VERB_PHR)) if
        comverbexp(COM_VERB_EXP),
        conn(CONN),
        comverbphr(COM_VERB_PHR),!.

comverbphr(cvp2(COM_VERB_EXP)) if
        comverbexp(COM_VERB_EXP),!.
```

5.4.1 The Semantic Processor Registers

Some processor predicates have been designed to ‘push’ syntactic structures onto an environment register while other predicates have been designed to collect syntactic substructures by ‘popping’ the appropriate register stacks. Each register is a list composed of unique stack elements.

The following is a brief description of the stacks employed by the semantic processor.

Primary Registers

The primary stacks are used for fundamental gathering of primitive tokens. These stacks are later popped by other processor predicates.

The Adv Stack The adverb, appearing as either `adv1(String)` or `adv2(String)` compounds, when encountered by the adverb decomposition predicate is placed on the adverb stack. The adverb stack, a list of adverb compounds stored as `adv(AdverbList)` is popped and the new adverb compound is appended to the end of the list stack.

The Adj Stack An adjective, either `adj1(String)` or `adj2(Real)`, when encountered by an adjective decomposition predicate, is placed on an adjective stack. The adjective stack is composed of the modified noun tag and the adjective list.

The following structure represents an adjective stack:

`adj(String,AdjectiveList)`

Since it is generally assumed that adjectives precede the modified noun, the adjective stack initially appears with the `String` value set to `unknown`, later altered to the noun when finally encountered during decomposition. A new adjective is added to the `unknown` adjective stack by appending the compound to the end of the adjective list.

The P Stack The possible prepositions *into, onto, to, on, in* and *of*, when encountered by the preposition handler, are asserted onto a `p()` stack, appearing as a set of asserted PROLOG compounds e.g. `p("to")`.

5.4.2 Register Collection and Distribution

During the operation of the semantic processor the primary registers undergo an ebb and flow of assertions and retractions. The assertion phase of primary registers is straight forward and will not be elaborated. The retraction of these primary stacks and reassertion within secondary registers, however, requires explanation. This stack handling can be roughly divided between four operational types:

- A Verb Handler
- The Noun and Vector Handlers
- A Task Plan Handler
- A Conditional Handler

At the conclusion of the semantic processing a set of *job* stacks is the only record of the meaning. These stacks are redistributed into individual *command templates*, uniquely describing each command, by a Command Template Generator.

The following sections will briefly discuss the operation of these stack handlers and the template generator.

The Verb Handler

Verbs are modelled as the PROLOG compounds `verb1(String)` or `verb2(String)`. When the verb handler encounters a verb, the verb stack, a list of `v(String,AdverbList)` compounds stored as `action(VerbList)`, and adverb stack are retracted. The new verb and its associated adverb stack are incorporated into a new `v()` structure and added onto the VerbList in the `action()` stack. The `action()` structure is then reasserted.

An exception to this rule is the verb 'to be', handled independently in the Conditional handler.

The Noun and Vector Handlers

When a noun is encountered, a *type tag* is used to direct the noun to the correct handler. These type tags are assigned early in the decomposition phase by 'phrase level' decomposition predicates. The type tags employed are described in table 5.4:

Each type tag differentiates a noun stack. Depending on the type tag a noun may trigger collection of different primary stacks. For reasons discussed later each noun handler raises a flag called `lastnoun(TypeTag)` to record the evaluation of a noun.

A single noun handler manipulates *io* and *do* noun stacks. These tagged stacks are stored in the following PROLOG compound data structure: `frame(TypeTag,FrameList)`. The `TypeTag` may be either `io` or `do`. The `FrameList` is composed of a list of `f(Noun,AdjectiveList)` data structures where `Noun` is a string and `AdjectiveList` is a list of adjective compounds.

When an `io` or `do` tagged noun (a string) is passed to the noun handler, the similarly tagged frame stack and the unknown adjective stacks are retracted. The noun and adjective list are then combined into an `f()` structure. This structure is then added onto the head of the frame stack, the stack is reasserted and the `lastnoun()` flag `io` or `do` raised.

The `sub` type tag marks all declarative clause subjects, invoking the *subject* noun handler. In this noun handler, the existing subject stack, a list of subject data structures, each one appearing as: `s(String,Declaration,AdjectiveList)`, is retracted. The noun is combined with the declarative verb "is" and the noun's adjectivelist. The stack is then reasserted and the `lastnoun()` flag `sub` raised.

The `locative` type tag implies that the noun is the object of a locative preposition. The locative preposition is used as a modifier on a noun or as a complement within a declarative clause. Therefore it was decided to place locative prepositional phrases into the *preceding noun's adjective list.* The noun handlers, therefore, were designed to raise the `lastnoun()` flag, recording the last noun's type tag. This flag directs the retraction and reassertion of the last noun's adjective list.

When an object of a locative preposition is encountered, the last noun and

Table 5.4: The Type Tags employed by TSKMSTR's semantic processor.

Type Tag	Noun or Vector Type
io	indirect object
do	direct object
sub	subject
locative	object of the locative preposition
src	object of the source preposition
pth	object of the path preposition
gl	object of the goal preposition

adjective list is retracted. The preposition stack and unknown adjective stacks are popped and combined with the prepositional object to form the locative construct: `loc(Preposition,Object,AdjectiveList)`. This is then inserted into the last noun's AdjectiveList and the last noun construct reasserted. The `lastnoun ()` flag `locative`, is raised.

A special locative case concerns appositive or successive locative prepositional phrases. Bennett [52] points out that :

> " Successive appositive expressions provide additional information about a single location"

For the semantic processor to provide for this interpretation requires that successive locative phrases be installed recursively in the AdjectiveList of the previous locative phrase e.g.*on the floor in the closet* becomes:

`loc(on,floor,[loc(in,closet,[])])`

When a non-locative preposition is encountered, the type tag, one of either *src, pth* or *gl*, identifies the noun with a `p_plan(TagType,PlanList)` register composed of a list of PROLOG compound prepositional structures of type `plan(Preposition,Object,AdjectiveList)`. The `p_plan`, `p()`, and `adj()` stacks are retracted. The preposition, noun and AdjectiveList are incorporated into a `plan()` structure, added to the `p_plan()` stack and reasserted. Finally, the `lastnoun()` flag, the TagType, is raised.

Since the *vector* structure, a compound containing either one, two, three, or seven real numbers, can be used in place of a noun, an identical set of procedures to those above handles vector references. The only exception being that locative vector structures cannot be used successively. As above, each vector may be asserted onto a stack of the appropriate `TypeTag`, though commonly vectors tend to occur in the `p_plan` registers.

The Task Plan Handler

A TaskPlan is a data structure designed to reflect a transition from one state to another, containing *source*, *path*, and *goal* fields. This data structure is presented as a triple of `plan()` lists within a `tskpln()` PROLOG compound structure. For example the transition *from the feeder through the washer to the drier* is modelled as:

```
tskpln([plan(from,feeder[])],[plan(through,washer,[])],[plan(to,drier,[])])
```

The use of lists for each field allows for multiple sources, paths, or goals within a single command clause. Each clause has one `tskpln()` structure, compound sentences therefore generate a stack of these structures stored in the `taskplan()` stack.

After the evaluation of a command clause, the Task Plan Handler retracts the `taskplan()` stack and the appropriate typed `p_plan()` stacks, constructs a `tskpln()` list element, and then reasserts the `taskplan()` stack.

The Conditional Handler

The Conditional handler builds data structures that describe some boolean condition that must be proven. These conditions are expressed within declarative clauses or clauses with the main verb 'to be'. This verb has unusual characteristics. The 'object' of a declarative clause can be either a *predicate nominative*, renaming the noun or a *predicate adjective*, an adjective or adjectival phrase modifying the subject.

The current implementation of the semantic processor supports the predicate adjective interpretation of the declarative clause. The subject construct, the `s()` compound described above, forms a state description for later proof by TSKMSTR.

The condition handler inspects an adverb list for the adverb "not" and, if found, the `Declarative` field in the `s()` structure is assigned to `not`. These structures are stored in a `chk()` checklist stack.

The Command Template Generator

Upon completion of semantic processing the noun, verb, and task plan stacks are collected and distributed to create command templates. The command template is a convention used to model the core meaning of an input command. The command template composition is detailed in figure 5.17. As the figure shows, the template is divided into four sections: the verb, an indirect object, a direct object and a *task plan*. The reasoning behind the selection of these structures follows:

The Verb Field : The verb field is used primarily as an identifier for the rest of the command template, analogous to a procedure name.

The Indirect Object : The indirect object acts as a catchment for nouns acting *like* indirect objects. For example:

Move the box 5 inches up.

This sentence demonstrates the syntactic complexity of even a simple command. The noun *box* in an initial analysis might be termed a direct object, a notion reinforced by the permutation:

Movethe box up 5 inches.

which leads the human parser to believe *up* is a preposition. This is not so, *up* is a *particle* or a preposition that is part of the verb (The verb is *move-up*). In this instance *inches* can be interpreted as the direct object and *box*, the indirect ob-

```
COMMAND      →  com(VERB,IO,DO,TASKPLAN)
VERB         →  v(STRING,ADV-LIST)
IO           →  f(OBJECT,ADJ-LIST) | nil
DO           →  f(OBJECT,ADJ-LIST) | nil
TASKPLAN     →  tskpln(SOURCE-LIST,PATH-LIST,GOAL-LIST)
SOURCE-LIST  →  list of SOURCEs | nil
PATH-LIST    →  list of PATHs | nil
GOAL-LIST    →  list of GOALs | nil
SOURCE       →  src(from,OBJECT,ADJ-LIST) |
                src1(from,VECTOR)
PATH         →  pth(PATH-PREP,OBJECT,ADJ-LIST) |
                pth1(PATH-PREP,VECTOR)
GOAL         →  gl(GOAL-PREP,OBJECT,ADJ-LIST) |
                gl1(GOAL-PREP,VECTOR)
OBJECT       →  coordinate frame label STRING
ADJ-LIST     →  list of ADJ
ADV-LIST     →  list of ADV
ADJ          →  loc(LOC-PREP,OBJECT,ADJ-LIST) |
                adj1(STRING) |
                adj2(REAL) | nil
VECTOR       →  dim7(REAL,REAL,REAL,REAL,REAL,REAL,REAL) |
                dim3(REAL,REAL,REAL) |
                dim2(REAL,REAL) |
                dim1(REAL)
ADV          →  STRING
LOC-PREP     →  in | on | at | of
PATH-PREP    →  through | via
GOAL-PREP    →  into | onto | to
```

Figure 5.17: The structure of *command template* used in the Semantic Processor

ject [1] TSKMSTR views particles as adverbs (though this is strictly not correct as demonstrated by: *Up move the box 5 inches* a legal parse for TSKMSTR).

The Direct Object The bulk of nouns appear in direct object field and usually represent the object of an action.

The Task Plan As explained earlier the task plan details a state transition, presumably directed at the Direct Object.

Upon completion of semantic processing, four remaining stacks, `action(VerbList)`, `frame(io,IOList)`, `frame(do,DOList)`, and `taskplan(TaskPlanList)`, must be redistributed into meaningful command templates through the use of a 'stack distribution convention'.

For example the command:

Clean and dry the floor and the window.

requires special distribution due to the compound verbs and direct objects. To an experienced human listener the above command clearly implies that both the floor and window be both cleaned and dried, though there is no implied order of events. The state of the stacks at the termination of the semantic analysis is shown below:

```
VERB              = actions([v("dry",[]),v("clean",[])])
INDIRECT OBJECT   = frame(io,[])
DIRECT OBJECT     = frame(do,[f("window",[]),f("floor",[])])
TASKPLAN          = taskplan([])
```

---

[1] This is not strictly correct, since the particle *up* is probably the remains of the preposition *up by* which would indeed make *inches* the object of the preposition. However, Roberts [58] suggests that of four tests for indirect objects, only the consecutive placement of two objects is reliable. This is the case in Bock's grammar. TSKMSTR therefore views *box* and *inches* as the indirect and direct objects respectively.

A stack distribution convention must reliably convey the correct semantics implied by the command. TSKMSTR's distribution convention orders command execution such that all operations on one object are completed before continuing to another object i.e.:

```
com(v("clean",[]),nil,f("floor",[]),nil)
com(v("dry",[]),nil,f("floor",[]),nil)
com(v("clean",[]),nil,f("window",[]),nil)
com(v("dry",[]),nil,f("window",[]),nil)
```

These commands are then placed into a command list. Though this method is not ideal (intelligent task queuing or scheduling is not provided), it does provide a simple, reliable, and safe task execution.

For multiple objects and task plans such as:

Move the jug and pot *to the glazer, to the drier and to the kiln.*

All object/taskplan permutations will be generated one object at a time in order of the task plans' appearance in the command.

The following is a complete example of a semantic processing session. Though it does not detail all the processors functions, hopefully it will provide the reader with a sense of the procedure.

5.4.3 An Example

For the parsed example in figure 5.16, *Completely affix the peg to the hole*, the semantic processor operates as follows:

- the adverb, *completely*, is isolated in the *adverb* decomposition predicate and asserted into an `adverb` register stack. The stack is a PROLOG list of adverb strings e.g

  `["completely"]`

- the verb, *affix*, is isolated in the *verb* decomposition predicate. Any adverbs appearing in the `adverb` stack are popped and combined with the verb in a record structure which, in turn, is pushed onto an `action` register stack. The `action` stack is a PROLOG list of action records composed of an action word and its modifier list e.g.

  ```
  [v("affix",["completely"])]
  ```

- *peg*, is type tagged `do` in the direct object decomposition predicate as a direct object and passed to the `io/do` noun phrase handler. All adjectives in the `adj()` stack are popped, incorporated with the noun into an `f()` structure and pushed onto a tagged frame stack. The stack is a PROLOG record composed of a tag and list of frame structures e.g.

  ```
  frame("do",[f("peg",[loc("on","peg"[])])])
  ```

  note that in this example the adjective list is empty or `[]`.

- the prepositional phrase *to the hole* is tagged as a 'goal' phrase, `gl`, implying some target position. The preposition 'to' is pushed onto the `p()` preposition stack and *hole* is passed to the preposition object handler. This handler pops the `p()` and `adj()` stacks, combining them with the noun to form a `plan()` structure that is pushed onto the `p_plan` stack e.g.

  ```
  p_plan("gl",[plan("to","hole",[loc("on","hole"[])])])
  ```

  Since no adjectives modify *hole*, the adjective list is empty, [].

Once the clause elements have been decomposed successfully the `action`, `frame`, and `plan` stacks are popped and combined into a *command template* by a command assembly function. The above example becomes:

```
com(
```

```
v("affix",["completely"]),
nil,
f("peg",[loc("on","peg",[])]),
tskpln(nil,nil,[p_plan("to","hole",[loc("on","hole",[])])])
)
```

5.5 The Command Interpreter

Once an English command has been parsed and converted into a command template and placed into a command list, the interpreter executes matching library routines. This requires a formalism for the description and execution of tasks. The following sections will briefly discuss the representation and performance of TSKMSTR commands.

TSKMSTR's task descriptions are designed to form a hierarchy of library tasks. Each task can be described as a list of subtasks. These subtasks in turn can be further expanded into more subtasks until the original task is decomposed into a list of executable 'primitive' tasks, described in section 5.6.

Each task definition is a hierarchical structure resembling a Horn Clause, similar to the structures developed in [17] and [21].

Each task definition head is a characteristic command template that is verified through the clause body, also composed of command templates. Any command template can be expressed, through Horn clauses, as a combination of the primitive templates. To accomplish this, each command template in the body is expressed in terms of previously taught routines, until the verification of templates produces a body of primitives that must be proven true. Each Horn clause is executed in a top down, depth first manner, identical to a PROLOG clause (see [39]).

Since tasks were to be created and run during TSKMSTR execution, a means of building and executing PROLOG-like programs within TSKMSTR (itself a PROLOG program) was required. TSKMSTR was written in a compiled PROLOG thus preventing the creation of new PROLOG clauses at run-time. The solution was to employ a *meta*-PROLOG, a meta-language used for the description of languages. In this case a PROLOG program was employed to model the action of PROLOG, in effect recreating PROLOG on top of PROLOG. A portion of an off-the-shelf meta-PROLOG [40], was modified and incorporated into TSKMSTR, providing the necessary PROLOG-like creation and execution of task meta-predicates.

The command templates become user defined meta-predicates, while the primitive templates became true PROLOG library and work cell function calls.

The PROLOG Interpreter and Translation Utilities

The PROLOG interpreter models a HORN clause with a simple but effective compound structure:

`cmp(":-",[Head,Body]).`

In fact, the meta-PROLOG `cmp()` structure models predicates and compound data structures in addition to the clause structure above. This meta-compound is generically represented by the following structure:

`cmp(Operator,ParameterList)`

`Operator` may be an operator, such as the 'if' symbol above, the 'and' symbol ',', the 'or' symbol ';', as well as the full range of arithmetic symbols. The operator may also be a predicate or compound name. The ParameterList is a list of parameters that can be any one of the following types:

- a `MetaVariable = var(String)`
- a `MetaString or atom = atom(String)`

- a `MetaReal = real(Real)`
- a `MetaList = list(Parameter,MetaList) or nill (for [])`
- a `MetaCompound = cmp(Operator,ParameterList)`

The following true–PROLOG clause:

```
echo_readln(STRING) :- readln(STRING), write(STRING).
```

therefore becomes the meta–clause:

```
cmp(":-",
    [
     cmp("echo_readln",[var(STRING)]),
     cmp(",",[
             cmp("readln",[var(STRING)]),
             cmp("write",[var(STRING)])
            ])
    ]).
```

Briefly, the interpreter's modifications encompassed the replacement of PROLOG standard calls (such as `read()`, `write()`, `sin()`, and `cos()`) with calls to work cell processes (such as `move()`, open(), and find()) and the removal of the interpreter's run–time environment and utilities (such as windows and file I/O).

Translation Utilities

A set of Translation Utilities converted a command template from a real PROLOG data structure to a meta–PROLOG predicate call. The *forward* translation phase (from true to meta–PROLOG) simply decomposed each command template into its constituent elements and converted them into meta–PROLOG parameter equivalents described above.

During the forward translation phase a series of tests inspected the adjectival lists in the `f()` structure for correct locative specification. The forward translation stage allows the user to correct grossly erroneous `loc()` spatial references at a stage that permits the task to continue without failure (a similar error later, in the world model, would cause an irretrievable global failure of the task).

Any position in the work cell is specified by a noun and its locative prepositional adjectives. The world model requires a **feature/part** pair to uniquely identify a point in space. A **feature** is defined as a coordinate frame of a **part**. The **part** field indicates the name of the host object. TSKMSTR expects nouns to represent either features or parts. Tests must be performed to discover the intended meaning of a given noun since users may refer to a feature or part in isolation. These tests will produce a **feature/part** pair based on an examination of the world model.

Each spatial reference undergoes the following four tests comparing the world model against the locative adjectives modifying the noun.

- If a **part** has no locative modifiers, the interpreter inspects the model for a **part**. If successful the full locative description of *noun* is supplied by the interpreter. For example *the box* initially interpreted as :`f(box,[])` becomes `f(box,[loc(of,box,[])])`. Otherwise the test fails.

- If a **part** is modified by successive locative phrases, the test inspects the model for agreement. If successful the execution continues. For example *the box on the top of the pallet* initially interpreted as `f(box,[])` becomes `f(box,[loc(of,box,[loc(on,top,[loc(of,pallet,[])])])])` . Otherwise the test fails.

- If a **feature** is modified by a single locative phrase, a test inspects the model for agreement. If successful the execution continues. For example *the top of the box* becomes `f(box,[loc(on,pallet,[])])`. Otherwise the test fails.
- If a **feature** is modified by successive locative phrases, a test inspects the model for agreement. If successful the execution continues. For example *the bottom of the box on the top of the pallet.* is initially interpreted :

  ```
  f(bottom,[loc(of,box,[loc(on,top,[loc(of,pallet,[])])])])
  ```

  Otherwise the test fails.

A failure in any of these tests results in the inspection of the dialogue stack, a list of recently discussed frames. Failing a useful match in the dialogue, a query session is invoked in which the user must specify the intended frame from the world model. Once these tests have been successfully navigated the correctly formed command template is converted into the meta–equivalent.

Prior to execution a final test is performed on the meta–command template that examines the command library for the existence of the meta–predicate's name (the verb) and parameterlist. A failure to find a match invokes a Teach Mode, described in the section 5.5.1.

5.5.1 Teach Mode

If a command is unrecognized during a dialogue the teach mode is invoked. The teach mode is a means of describing a command unrecognized by the interpreter in terms of recognizable commands. In effect, the teach procedure is an English explanation of an unknown command.

Prior to entry into the teach mode proper, the user is asked a series of questions about the nature of the noun and adjective values in the unknown command. The purpose of this 'interrogation' is to determine possible variable terms in the command. The user is asked to decide whether a given term is unique to this command or is generally applicable to other situations.

This query 'engine' works through the *io*, *do*, and *taskplan* fields asking for the generality of each term. If a term is selected as 'general' then a temporary variable flag is raised, noting that all instances of the term must be interpreted as a variable.

During the teach dialogue, TSKMSTR constructs command templates just as it would in the interpreter, but does not actually execute the command. In the forward translation in the teach phase, nouns and adjectives are compared against the variables flags raised in the query session . If a match is found, terms that would normally be converted directly to their meta–equivalent are, instead, transformed into the `var()` compound described above. For example: if in the command *Grasp the box.*, *box* is a variable, then the *do* field, `f(box,[])`, becomes

```
cmp("f",[var(BOX),var(BOX_LOC)])
```

instead of the usual:

```
cmp("f",[atom(box),nill])
```

Note that if an `f()`,`plan()` or `loc()` structure contains a variable adjective list, the adjectival list may also become a variable. For example in *Grasp the handle on the box*, if *box* is still a variable, `f(handle,[loc(on,box,[])])`, becomes

```
cmp("f",
    [atom(handle),
     list(cmp("loc",[atom(on),
                     var(BOX),
                     var(BOX\_LOC)]
             ),
          nill)
    ]
   )
```

instead of:

```
cmp("f",
    [atom(handle),
     list(cmp("loc",[atom(on),
                     atom(box),
                     nill]
             ),
          nill)
    ]
   )
```

As the session progresses, the 'teach' meta-command templates are incorporated into the meta-clause structure described earlier . Upon entry of the 'end' keyword the session finishes and the command is executed.

If, during the teach phase, a command is issued that does not exist in the library, the teach mode will be immediately reinvoked. This recursive reinvocation or 'descent' of the teach mode will continue *ad infinitum* until the user provides explanations that are in terms of existing 'lower level' library commands. As each teach session is completed, the user 'ascends' once again to the next level of instruction, thus allowing the composition of very high level commands from the top down.

5.5.2 Unsupported Legal Syntactic Constructs

Though the parser and semantic processor are able to generate descriptive constructs for both conditionals and simple adjectives, the command interpreter does not currently support them.

Conditionals are not supported for two reasons, the first is succinctly stated by Lieberman and Wesley [8]:

> " Although severe in a general programming sense, it seems that apart from error recovery there is little need for complicated branching paths in an assembly program, and in practice the user may not be affected."

In other words, an assembly language does not need extensive branching mechanisms like those implied by English conditional dependent clauses. Secondly, the horn clause theorem proving approach renders these clauses largely unnecessary by ensuring that conditions for the assembly are correct. For example the command: *If the box is on the pallet, grasp the box.* would be treated identically to the command: *Grasp the box on the pallet.* since the inclusion of the adjectival locative phrase *on the pallet* forces TSKMSTR to search for a relationship between the pallet and the box.

The removal of simple adjectives from the semantic evaluation is a simplification of the prototype system and in no way precludes their treatment at a later time. The current decision lay in that most adjectives can be 'built in' to the frame name (e.g. square hole becomes square_hole), with little loss of meaning or flexibility. Adjectives of comparison, such as *larger* and *longer*, however, require a procedural analysis of the world beyond the descriptive capacity of the current world model and are, therefore, not supported by this prototype NLI.

5.6 World Model, Kinematic Engine and Task Primitives

5.6.1 World Model and Kinematic Engine

The world model, also written in PROLOG, is oriented towards assembly programming and has been designed to be hidden and self-maintained, requiring minimal user attention or kinematic knowledge. It employs a graph approach of the world similar to AL.

In the TSKMSTR Kinematic Inference Engine (KIE), assigning the position of the gripper or **tool**, modelled as a coordinate frame, to some target location, also described by a frame, is equivalent to moving the robot, i.e. a robot motion can be described as the conjunction of two coordinate frames. In TSKMSTR, frame conjunctions come in three varieties: *into, onto,* and *to*. These -to relations become the natural language object (or frame) interrelationships: *in, on,* and *at*. In the TSKMSTR world model the z–axis of each coordinate frame is assumed to be either perpendicular to an object surface or aligned with the axis of a hole. The conjunctions *in* and *on* are identically defined and oppose the two frames' z–axes, while *at*, an identity matrix, aligns the two frames' z–axes exactly.

An assembly graph representation has been adopted to capture these interrelationships and to describe the location of assemblies in the world. Each *node* in the graph represents a coordinate frame and each *arc*, a homogeneous transformation. An object with a group of n subcomponents called *features* is described by a group of n arcs and $n + 1$ nodes as in figure 2.2.

A node in AL is usually represented as a record structure. A disadvantage of this PASCAL–like notation is that the record is a typed variable. An alternative approach, freeing the user from this rigid typecasting, was the use of a PROLOG record structure, called a *compound*, containing a feature/part pair, e.g. a hole in a plate becomes `node(hole1,plate)` where `hole1` is a position or feature and `plate` a reference to a part. The advantage is that both the feature and part fields are variables, entirely unlike the pure record equivalent e.g.`plate.hole[1]`. This allows the world model to inspect and *create* feature/part pairs.

In the graph, any object node can be attached to any other object node through the use of a conjunction transformation arc. In a prototypical form, an arc might be described by the following compound structure.

```
arc(
    node(plate,plate),
    ht(v(1,0,0),v(0,1,0),v(0,0,1),v(1,3,0)),
    node(hole1,plate)
   )
```

Where `ht()` and `v()` are homogeneous transformations and vectors respectively and `node(plate,plate)` is the reference coordinate frame on the plate. An assembly of two objects, therefore, is composed of two object features connected to one another through an *in, on* or *at* conjunction transformation. The assemblies become more complex as these conjunction transformations are used to join several parts.

The location of any node relative to another node can be derived by finding a path of arcs between the two nodes. KIE employs a breadth first search method to build these arc paths. In an assembly graph model, a single node may be reached by multiple paths, some of which may contain 'cycles' or paths through previously traversed nodes. KIE's search method ensures that cyclic routes, common in assemblies, will not be generated and guarantees that the derived path will always be the shortest.

Though more efficient PROLOG breadth first methods are possible, the prototype KIE breadth first search algorithm proceeds as follows. A list of possible paths through the graph is generated as the search progresses. Each path is described by a list of constituent nodes. New nodes are always added to the front or *head* of this node list. The procedure compares the head of a path list to the target node. If the head is *not* the target node, the head's neighbouring nodes are collected and each new node added onto a copy of the path list. For example: if the head node of a path list has n neighbouring nodes, the path will be replaced by n copies of itself, each with an additional node at the head. These "expanded" paths are *appended* or added onto the end of the list of paths. The next path in the list is then examined in the same manner through the use of a *recursive* predicate call.

This expansion and subsequent concatenation of paths results in the perpetual inspection and (further) expansion of paths in a breadth first manner. The following PROLOG code fragment presents a basic example of this method.

```
breadth_first(END,[[END|P]|_],[END|P]) if !.
breadth_firt(END,[[N|P]|OTHERS],S) if
        expand_nodes([N|P],NEW),
        append(OTHERS,NEW,PATHS),!,
        breadth_first(END,PATHS,S)
        or
        !,
        breadth_first(END,OTHERS,S).
```

Where `END` is the target node, `P` is some path, `N` is some node, `OTHERS` are alternate paths, `NEW` are new paths through `N`, `PATHS` is the new list of paths, and `S` is the solution path.

As mentioned above, assembly graphs often contain cycles or redundant paths through the graph. To avoid "going around in circles" the `expand_nodes()` function does not collect nodes that are already present in the current path. Since a single arc describes an attachment between two nodes, the collection of neighbouring nodes must be prepared to inspect both node fields in the `arc()` data structure. Once the path is determined, the product of the chain of homogeneous transformation arcs along the path expresses the location of the target node relative to the start node's coordinates.

It is worth noting that this PROLOG breadth first search employs both the common *recursion* PROLOG technique and PROLOG's powerful built-in unification mechanism to make the procedure compact, efficient, and simple to implement. [39].

The assignment of a **tool** frame,(i.e. a frame attached to the gripper), to a **target** frame (i.e. a frame not attached to the gripper) is equivalent to creating a new conjunction *arc* in the graph (analogous to a homogeneous transformation), extending from the **tool** node to the **target** node. The linking of these frames with an arc implies that the two frames are somehow spatially interdependent. As described earlier, this interdependency is not necessarily mutual. In order to reflect the asymmetry of spatial interdependence common in the real world, an AL–like *affixment* scheme has been employed. A pair of affixment flags declare the dependency between nodes linked by a conjunction arc. For example in figure 5.18:

> If block A is resting on block B, node A_{bottom} and node B_{top} are connected by an arc with the affixment flags, ${}_{A_{bottom}}F_{B_{top}}$ and ${}_{B_{top}}F_{A_{bottom}}$. Since block A is positionally dependent on block B (if block B moves so will block A) then ${}_{B_{top}}F_{A_{bottom}}$ = `fixed`. However, ${}_{A_{bottom}}F_{B_{top}}$ = `unfixed`, since the movement of block A will *not* influence block B.

In the normal movement of a **tool** frame to a **target** frame, the **tool** and **target** affixment flags are initially assigned to `unfixed` i.e. no permanent spatial interdependence is assumed. The **target** frame may be incorporated into the **tool** by fixing the **target** to the **tool**, equivalent to setting the **target** flag, ${}_{tool}F_{target}$, to `fixed`.

Subsequently, any motion in the **tool** frame is reflected in the **target** frame (and any other frames fixed to the **target**). Conversely a tool **part** can be attached to the target by *unfixing* the **part** frame from the rest of the **tool** and affixing the **part** frame to the **target** frame. *Unfixing* is equivalent to assigning the **part** affixment flag, ${}_{tool}F_{part}$, to `unfixed`.

When the **tool** assembly is moved by the manipulator all temporary relations between the **tool** and the world are broken by an `isolate_tool()` function. This is accomplished by removing all the `unfixed` arcs in the traversed tool–to–world direction with an acyclic breadth first search method similar to the one described earlier. During the `expand_nodes()` phase all `unfixed` flags in the tool–to–world direction are *retracted* or deleted from memory. The acyclic search saves the remaining `unfixed` arcs in the world–to–tool direction from being traversed and retracted. The net result is that all unfixed connections the tool has with the world are severed, while preserving all the fixed connections to the tool. For example in figure 5.18: If the gripper is raising block B.

> On the outward traverse of the arc between block B and the world the search will encounter and eliminate the arc tagged by ${}_{B_{bottom}}F_{world}$, since ${}_{B_{bottom}}F_{world} =$ `unfixed`. On the outward traverse of the arc between Block B and A, the search will encounter and ignore flag ${}_{B_{top}}F_{A_{bottom}}$, which is `fixed`, and move on to Block A. At Block A the search method prevents the backwards traversal of the arc connecting Block A and B, thus the search avoids removing the flag ${}_{A_{bottom}}F_{B_{top}}$, which is *unfixed*, and retaining the positional interdependence between Blocks A and B.

The final data structure describes the arc between two nodes in the graph, each named with a feature/part pair. Since the acyclic search method prevents the pursuit of previously traversed nodes, the two affixment flags can be safely combined into a single arc (each node flagged with an affixment flag). Homogeneous transformations describe the change in location between the two nodes, transforming the world model into a *directed* graph, (the sense of the transformation is from left to right).

$$\mathrm{arc}(\mathrm{node}_1, {}_{node_1}F_{node_2}, {}^{node_1}T_{node_2}, {}_{node_2}F_{node_1}, \mathrm{node}_2)$$

An example:

```
arc(
```

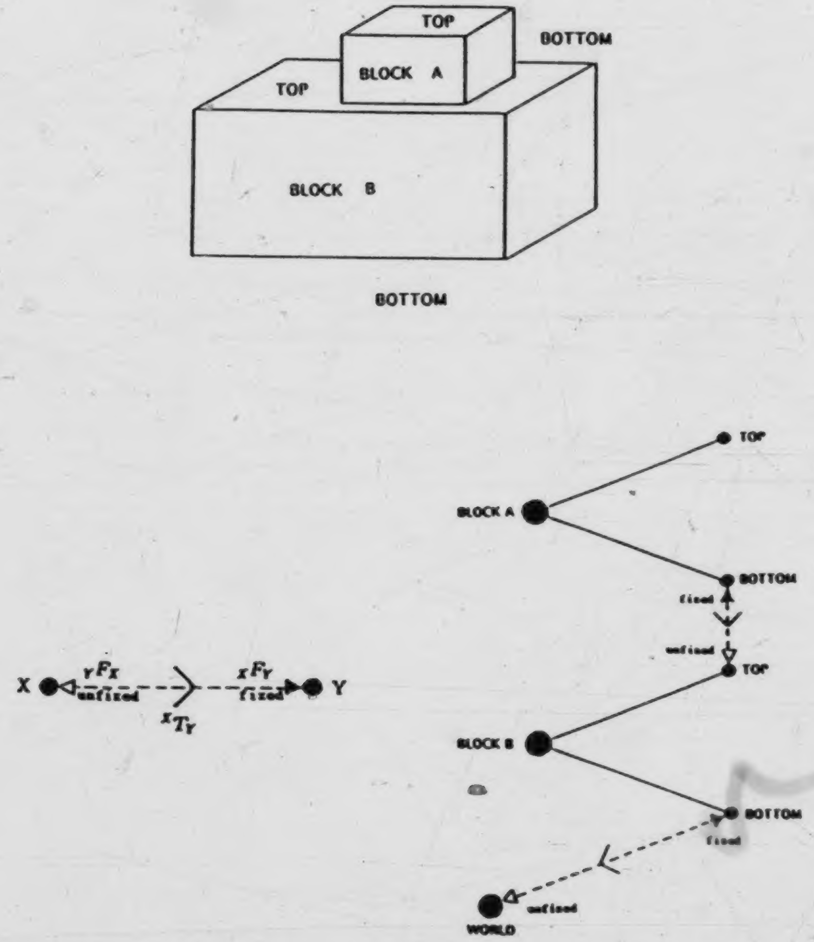

TOP
BOTTOM
BLOCK A
TOP
BLOCK B
BOTTOM
X
$_YF_X$
unfixed
^XT_Y
$_XF_Y$
fixed
Y
TOP
BLOCK A
BOTTOM
fixed
unfixed
TOP
BLOCK B
BOTTOM
fixed
unfixed
WORLD

Figure 5.18: KIE's affixment notation for a stack of blocks.

```
    node(plate,plate),
    fixed,
    t(hole,ht(v(1,0,0),v(0,1,0),v(0,0,1),v(1,3,0))),
    fixed,
    node(hole,plate)
  )
```

Each transformation is labelled to allow the construction of symbolic task equations. Note that since this describes the `hole` as a feature of the `plate`, the affixment flags are both set to `fixed`. Regardless of the direction of the tool isolation procedure, the arc will be preserved. This data structure in conjunction with the breadth first search mechanism effectively describes assemblies within TSKMSTR's world model, See figure 5.19.

Finally, if the search is unsuccessful, the vision system is engaged, providing the identity, location, and orientation of objects in the camera's field of view. Any objects in view are subsequently incorporated into the world model by linking the object reference node, `node(object,object)`, on the bottom surface of the object, to the world node. In keeping with the world model convention, the relation between the world node and the object surface frame is modelled through a world *feature*, a temporary intermediate point located on a world surface. This point is created by the kinematic engine beneath the object on the world surface, the standard *at* conjunction transform is then used to link the nodes e.g.: If a disk is found in the view area, the relation between the world and the disk nodes can be summarized by two `arc()` expressions.

```
arc(node(world,world),fixed,<some transform>,fixed,node(W.disk,world)
 arc(node(W.disk,world),unfixed,<AT transform>,fixed,node(disk,disk))
```

The expression `<some transform>`, the position of the new object in the world, is determined through a simple matrix computation relating the position of the object as the camera matrix product discussed earlier and fully described in Paul [22]. The

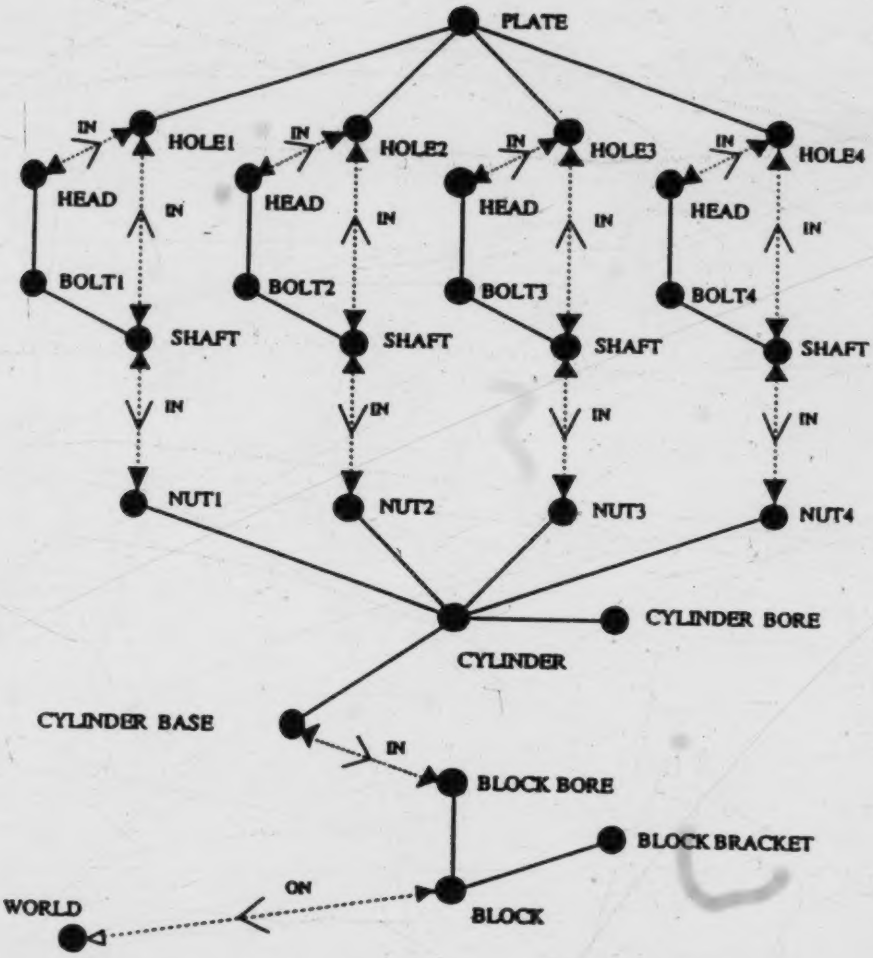


PLATE
HOLE1
HOLE2
HOLE3
HOLE4
IN
HEAD
BOLT1
BOLT2
BOLT3
BOLT4
SHAFT
NUT1
NUT2
NUT3
NUT4
CYLINDER
CYLINDER BORE
CYLINDER BASE
BLOCK BORE
BLOCK BRACKET
BLOCK
ON
WORLD

Figure 5.19: A complete Assembly Graph as modelled in KIE for the objects shown in figure 2.2.

search algorithm is then restarted. A further failure to find the errant frame will fail the entire procedure.

5.6.2 Task Primitives

Once the positions of the frames have been determined the task primitives are free to execute robot manoeuvres. In general a task primitive is some manipulation of a frame in the work cell. The current command primitives are ten low level calls to the work cell. See table 5.5. In essence these commands may be summarized as follows:

- Move a frame to another frame or to a unique XYZRPY (inches,degrees).

  $$T_n \text{ TOOL} = \text{ TARGET} \quad \text{or} \quad T_n \text{ TOOL} = \text{XYZRPY}$$

  Typical usage:
  Move the gripper to the handle on the box.
  or
  Move the handle to xyzrpyg(17,0,0,0,180,0,0).

- Move (Translate) a frame to a position relative to its current position in work cell coordinates (inches or centimetres).

  $$T_{n_{new}} = \mathbf{Trans}(x, y, z)\, T_{n_{old}}$$

  Typical usage:
  Move the handle on the box up 5 inches.

- Move the robot to a joint position (hundredths of a degree). Typical usage: *Move the robot to position(9000,0,0,9000,0,0,0).*

Table 5.5: A list of the TSKMSTR primitive work cell functions.

Verb	Indirect Object	Direct Object	Taskplan
MOVE	—	*frame*	*frame(s)* | xyzrpy
MOVE	*frame*	*units*	—
MOVE	—	robot	position
AFFIX	—	*frame*	*frame*
UNFIX	—	*frame*	*frame*
FIND	—	*frame*	—
ASSIGN	—	*frame*	xyzrpy
ROLL	*frame*	*units*	—
PITCH	*frame*	*units*	—
YAW	*frame*	*units*	—
OPEN	gripper	*units*	—
CLOSE	gripper	*units*	—

- Affix a frame to another frame. Typical usage:

  Affix the gripper to the handle on the box.

- Unfix a frame from another frame. Typical usage:

  Unfix the gripper from the handle on the box.

- Find the position of a frame. Typical usage:

  Find the handle on the box.

- Assign a frame label to a unique XYZRPY (inches,degrees). Typical usage:

  Assign handle on the box to xyzrpyg(17,0,0,0,180,0,0).

- Roll a frame to a position relative to its current position in tool coordinates (degrees or radians).

  $$T_{n_{new}} = T_{n_{old}} \mathbf{Rot}(z, \theta)$$

  Typical usage:

  Roll the gripper 30 degrees.

- Pitch a frame to a position relative to its current position in tool coordinates (degrees or radians).

  $$T_{n_{new}} = T_{n_{old}} \mathbf{Rot}(y, \theta)$$

  Typical usage:

  Pitch the handle on the box 17 degrees.

- Yaw a frame to a position relative to its current position in tool coordinates (degrees or radians).

  $$T_{n_{new}} = T_{n_{old}} \mathbf{Rot}(x, \theta)$$

  Typical usage:

  Yaw the box -20 degrees.

- Open the gripper a specific number of units (dimensionless). Typical usage: *Open the gripper 100 units.*
- Close the gripper a specific number of units (dimensionless). Typical usage: *Close the gripper 20 units.*

5.7 Communication and Support Utilities

Since the robot controller sends and receives commands over an RS-232c serial line, an interrupt driven communication utility [41] was employed to handle the 9600 baud robot and vision system message traffic.

PROLOG, while being an effective symbolic manipulation language, is a poor numeric processor, having no facilities for multidimensional array structures. Therefore homogeneous transformation inversion, matrix multiplication, and both the forward and inverse kinematic solutions for the Excalibur were implemented in 'C' and incorporated into the TSKMSTR environment. As mentioned previously the homogeneous transformations appear as PROLOG compound structures composed of four subcompounds each containing three real numbers representing a vector in space. The ability to represent matrix equations as PROLOG lists simplifies the evaluation of kinematic chains through the use of recursive calls to the 'C' matrix multiplication utilities.

The following chapter will discuss the performance of the system and present conclusions and recommendations for further research into Natural Language Interfaces for robotic work cells.

Chapter 6

TSKMSTR Evaluation, Recommendations and Conclusions

In this chapter the efficiency and function of each TSKMSTR module will be discussed. Recommendations will then be made regarding future modifications to this software and research into other Interface issues for robotic work cells.

6.1 Natural Language Interface Performance and Evaluation

An unfortunate characteristic of both natural language processing and PROLOG is that traditional methods of software evaluation such as execution time or floating point precision benchmarks do not apply. Both disciplines employ search techniques whose execution time is often highly dependent on the physical order of the data in memory as in the case of a PROLOG database search. Depth first parsing is particularly vulnerable to this order of parse predicates. It is a simple exercise to develop sentence structures that maximize failure and backtracking (thus increasing execution time) once the order of parse predicates has been determined.

Though in a global sense execution time is important, i.e. the general algorithm must be reasonably fast, natural language processing speed is difficult to measure and benchmark. More so when real world processes such as sensor I/O and robot motion are included. The following discussion will, therefore, concentrate on code clarity and ease of modification and refrain from using execution speed as an evaluation datum.

6.1.1 Scanner and Parser Evaluation

Given that both modules operate within the bounds of an unoptimized syntactic grammar, the scanner and parser are fast and efficient. A typical scan and parse on the 80386 is completed in less than one half second, insignificant compared to I/O and robot motion time.

A disadvantage of the current parser is that only one parse, the first generated, is produced. Maas and Suppes' [21] system developed a group of legal parse structures from which the semantic processor evaluated only the *best* or most suitable parse. The current construction of the parser limits the generation of multiple parses for a single string. In PROLOG, the only means of generating different parses produced by a depth first search is through invoking failure and subsequent backtracking through the parse tree. The current parser limits this backtracking, through frequent use of the cut, '!', operator, for good reason. Without the liberal use of cuts the parser would soon overflow the backtracking stack, a likely situation with this large grammar. Stack overflow might be avoided, however, through the combination of bottom–up techniques with the existing top–down algorithm, the resultant 'shallower' search tree would reduce the size of the stack committed to backtracking.

Despite the apparent effectiveness of the code, the parser invariably repeats discoveries of legal constructs during the depth first search. This, too, could be avoided through a combined bottom–up/top–down parsing strategy.

These frailties aside, the parser generator seems to create relatively clear and compact PROLOG code.

6.1.2 Semantic Processor Evaluation

Like the scanner and parser, the semantic processor dominates the performance of both the executable and teach interface modes.

From the programming standpoint, the semantic processor is fairly straightforward PROLOG. The processing logic is essentially limited to simple slot-filling, with very little 'traditional' semantic evaluation like that found in [21]. The semantic processors chief advantage is the ease of modification and prototyping through the use of asserted 'register' stacks. While the frequent assertion of predicates is not elegant PROLOG, the technique enables for rapid prototyping and modification. Unfortunately this technique also results in relatively opaque PROLOG and is therefore considered poor PROLOG practice. The preferred method and PROLOG's greatest asset, *unification*, has not been fully embraced in this prototype semantic module. Without question, there is ample opportunity for the application of unification techniques throughout the semantic module. Though the processor is not slow, the efficiency, speed, and size of the code could be greatly affected by tighter PROLOG programming techniques.

Furthermore the semantic processing system, with appropriate unification programming techniques, could be built into the modified parser, ultimately producing multiple *semantic* interpretations of a command string and reducing code size (therefore increasing the available stack size) and execution time considerably.

The Command Template

The choice of a command template system for task representation is based on TSKMSTR's teaching methodology. The teaching philosophy adopted for TSKMSTR is similar to Bock's LOROB and MIDROB programming method, the difference being that the semantics of high level commands are produced through lower level English command sets instead of abstract BASIC–like routines. Through this mechanism TSKMSTR effectively shifts the responsibility of semantic processing onto the user who can precisely define his or her own semantic values for an English command string. A further advantage to this method, when combined with a PROLOG interpreter, is that the tasks can be generalized based on the user's requirements. An obvious danger of this is that poorly generalized commands may become little better than aptly named macros, unique to a limited set of circumstances and bearing only surface resemblance to the real English semantics. Careful generalization of task descriptions, however, should allow the user to develop a set of routines applicable over a wide variety of conditions.

Whether TSKMSTR's command template is optimal is difficult to determine, though some syntactic and semantic arguments justify the use of the verb, object and source/path/goal combination.

The semantics of assembly are primarily physical. Without a physical means of representing assembly semantics to an NLI, a process pursued by [18] with some success, there is no general way to relate the precise semantics of robot motion to the semantics of the English language. For example: the command *Assemble* is plainly meaningless without a direct object. Many verbs seem so dependent on objects as to reduce their semantic value to little better than a label. It is this dependence between verb and object that justifies the use of some command template method. Since there are no generic semantic constructs, and therefore procedures, for each verb in the English dictionary, semantic definitions can only be built through English explanations. Thus the use of the verb as an action procedure label may be justified.

The use of both indirect and direct objects in the command template is debatable. Certainly, the direct object is a very common construct indicating the object being manipulated or transformed. The indirect object has questionable semantic value, however. In fact, Roberts [58] suggests that the distinction between the direct and indirect objects should be dropped since both are substantives and indirect objects do not follow any single grammatical (or semantic) rule other than simple placement in the sentence. In TSKMSTR indirect objects are used rarely and at the work cell primitive level, only as catchments for successive objects not separated by coordinating conjunctions. For these reasons the indirect and direct objects have been maintained as command template elements.

Bennett's discussion on the spatial semantics of prepositions indicates that any transition may be completely described through the use of *source*, *path*, and *goal* phrases. Since both manipulation in space and the conjunction and assembly of two parts are essentially transitions, the combination of these three components in a single command template seems reasonable. It is worth noting that in our world model, *source* prepositional phrases are rarely used, (barring the `unfix` command), since the location of all world objects is known (or can be determined from the vision system). Positional specification through locative prepositional phrases is usually sufficient to uniquely locate any frame in the world e.g.:

move the box from the pallet into the truck.

may be stated as easily in:

move the box on the pallet into the truck.

and is more precise since *from* does not provide the specific conjunction relation between objects implied by *on*. Nevertheless, since there may be functions defined in terms of complete transitions, the *source* element has been retained.

Based on these arguments TSKMSTR's template seems to be a reasonable, though prototypical, semantic construct for the description of physical tasks.

As a final comment, it is worth noting that in Allen's discussion of Case

Grammars [56], the results of which are shown in figure 3.1, the terms THEME, BENEFICIARY, AT, FROM, and TO are entirely equivalent to the syntactic terms direct object, indirect object, locative, source and goal command template elements. This equivalency of case grammars to slots has been noted by Charniak [59]. Further, the command template bears considerable resemblance to the ATRANS structure found in CD analysis, particularly the ACTION, TO and FROM slots.

The current natural language processing modules account for approximately 87 kB of object code or 30% of TSKMSTR's source code. This may be considerably reduced by employing the recommendations outlined above.

6.1.3 The Command Interpreter

Executable Mode

The command interpreter, composed of translation utilities and the meta-PROLOG interpreter, is the chief mechanism through which tasks can be both executed and generalized. Compared to the other systems discussed earlier, use of this interpreter and associated generalization techniques represents a step forward in task representation. Though the current technique of plying the user with questions addresses only the surface structure of a meta-clause, the system does allow the user to generalize commands with some precision. Future systems should investigate how this technique might be employed to 'intelligently' modify user commands, perhaps through determining a command's intent rather than it's literal meaning.

Though the interpreter's execution time for a typical task is insignificant compared to the time required for manipulator motion and vision I/O, it is extremely consumptive of memory, since the stack is used to store the position of both the true and meta-PROLOG backtracking pointers. TSKMSTR's code size limits this stack space and, therefore, limits the 'depth' of calls that can be made to the meta-predicates. This severely restricts the complexity of TSKMSTR routines.

Fortunately the command interpreter is capable of writing user defined meta-clauses to external files as true PROLOG predicates. This provides two distinct advantages. Firstly, previously taught routines can be loaded into TSKMSTR through the **load** keyword and, secondly, complex or 'deep' meta clauses, built in TSKMSTR, can be stored in files and executed within the true-PROLOG compiler. With the appropriate task primitives and support utilities, these predicates can be run as traditional PROLOG routines with essentially no limitations on stack space and with the benefit of a faster execution time.

A wasteful but unavoidable drawback of the current interpreter is the forward and inverse translation between PROLOG data structures and the meta-PROLOG equivalents. This step could be avoided by generating meta-data structures directly within the semantic processor, a modification worth considering in the final system.

The frame tests mentioned in section 5.5 examine the nouns in a command string for correct specification and consult a simple dialogue stack for previously used nouns. While this dialogue and inspection system saves the user from continuously retyping common nouns, TSKMSTR lacks the facilities required to perform true ellipsis. Ellipsis occurs mainly between two clauses as in:

If the box is open, close *it*.

or within two distinct commands:

Grasp the box and place *it* on the pallet.

Since each command template is executed in isolation, elliptical references between commands should be resolved *before* the command interpreter is invoked, ideally within the semantic processor.

In total the command interpreter is approximately 44kB of object code. Further investigation is required to reduce the size of this interpreter and tailor its operation more closely to work cell requirements.

Teach Mode

The teach module builds meta-predicates with a small, simple and effective unification method. The query engine successfully isolates all potential variables outlined by the user, distinguishing between the noun and it's locative modifiers.

Though many syntactic substructures were not supported in this prototype, such as dependent clauses, the remaining language has been left largely intact with only a few exceptions. In particular, the TSKMSTR system has not employed the PROLOG 'or' operator, functionally equivalent to an 'else' statement in a procedural language. The omission of this operator stems partially from the potential difficulties of incorporating two English flow patterns into a top down teach method. An English 'or' symbol implies some alternative action, virtually identical to the PROLOG 'or' operator. Simple to implement in the Teach module, a divided flow pattern within an English procedure may nevertheless be confusing to the user. At each 'or' juncture the user would have to supply a top down definition of two alternative English procedures. This omission has not left TSKMSTR unable to handle alternative task descriptions, however. Alternative flow patterns can be easily modelled through the construction of two independent meta-clauses effectively simulating the Lieberman and Wesley's 'rare' `if-then-else` conditionals. Future versions should investigate the difficulties of using this operator within a top down English teach method.

The translation utilities and teach module account for approximately 52 kB of object code. The incorporation of the translation utilities into the semantic processing stage could reduce this size significantly. Though the current teach module is relatively compact, the system could profit from further optimization.

6.1.4 World Model Performance

In Executable Mode

The world model appears as a set of `arc()` PROLOG assertions. While this approach may seem an unnecessary complication given that the breadth first search appears to construct an AL-like 'calculator list', the `arc()` data structure enables a simpler evaluation of the world model than the AL model. The AL model imposes an artificial hierarchy of nodes with the 'parent' marking scheme, requiring a world model manager to maintain hierarchical relationships in the assembly. KIE's approach makes no assumptions about node hierarchies and concentrates only on node interdependence. This does not mean to say that hierarchies do not exist in this model, the presence of a viable path between nodes implies that such parental hierarchies must exist. The discovery of any hierarchical pattern, however, is an unimportant consequence of the search procedure. This saves the world model from repetitive updating and review as is the case in the AL world model's 'dynamic', 'validity', and 'parent' marking.

This method is far from perfect, however. The repetitive searches through the database are wasteful and the current search method can be further optimized. Artificial maintenance of the world model, while reduced to the removal of unfixed arcs by the `isolate_tool()` function, still remains functionally equivalent to AL's 'dynamic' marking. However, this prototype model seems conceptually simpler and operationally less complex than the AL model.

The current advantage of this system lies in its simple implementation and implicit behaviour. Though the model need not be implemented in PROLOG, the model algorithm lends itself to relatively simple PROLOG programming techniques such as depth first search, recursion, and backtracking. Despite PROLOG's bias towards depth first search, implementing breadth first techniques are fairly straightforward. It is, perhaps, worth noting that the entire prototype TSKMSTR kinematic engine is little more than 30 kB of object code and that the execution time on an 80386 PC is insignificant compared to the time required to send and receive sensor data and to execute robot motions in the work cell.

In Teach Mode

A serious failing of the teach mode is that the world model is not evaluated during the teach procedure. World model consultation is currently limited to the verification of object specification in the object library. A teach world model mimicking the executable world model is not built.

The reasons for this omission lie in the assumption that the teach phase occurs off line. Commands to the robot are not issued during this phase, and the world model, therefore, remains unaltered. Though it is certainly possible to build a teach model in TSKMSTR, it's application would be limited without being able to observe the progress of the assembly. Since there is little point in developing an 'off–line' teach environment that must be used on line, the next logical addition to the environment would be a graphical world model simulator. Given that the vision system/world model relationship allows for considerable discrepancy between the actual and simulated layout of a work cell, exact simulation of the work cell may not be necessary.

6.2 Hardware Evaluation

6.2.1 Manipulator Performance

The manipulator has been viewed as a simple output device to the TSKMSTR system and has been used for only simple assembly tasks. However, for real world applications some additions should be made to the robot's control routines. Linear motion facilities, provided by RSI [42], should be incorporated into the robot control software. Memory limitations have prevented their installation in this version.

The controller occasionally has difficulty processing incoming messages and, therefore, transmits error messages to the host computer reporting potential manipulator problems. Unfortunately, the arrival of these messages is unpredictable and TSKMSTR's 'fail safe' error handling approach amounts to failing the current task. A real world application must have robust error handling facilities capable of determining the cause and resolution of each error condition. This ability should be a mandatory component of the final version of TSKMSTR.

The world model relies on an accurate description of the environment, including the position of the manipulator. In the prototype software, TSKMSTR does not store the location of the robot and must request joint angles through the `MANIP` command issued to the serial port to guarantee accurate joint angles. An advantage of this system is that the `MANIP` request cycle ties the real world to the computer's world model. The controller command processor only acknowledges errors, legal commands producing only robot motion. When issuing a command stream, the host computer is ignorant of the actual command being processed by the controller at a given time. TSKMSTR relies on the `MANIP` cycle to resynchronize the command set, limiting the discrepancy between real and modelled worlds.

Though these drawbacks are rarely encountered, the time delay and occasional communication problems are a troublesome weakness in the current system. The manipulator position should be integrated into the world model, perhaps through another hardware link between the controller and the host computer, providing continuous values of the six joint angles.

6.2.2 Vision System Performance

VISYS is a good representative of more expensive, albeit faster, industrial binary vision systems. Industrial vision systems, orders of magnitude more expensive than VISYS, typically return a similar range of image data within a few seconds for complex pictures [48]. The VISYS system returns image data in approximately 5 seconds per blob in the image. A full frame with 6 objects returns complete object data in approximately 25 to 30 seconds depending on the perimeter pixel densities. Centroidal placement is repeatable to within a tenth of a millimetre. Positional accuracy, though not precisely determined, is well within the robot's $\pm 1.27mm$ positional accuracy.

Though the system is effective and inexpensive, the slow processing speed limits the number of times TSKMSTR can affordably consult VISYS. Currently the vision system is consulted only when absolutely necessary, after the world model search has failed, and is inadequate for real time processes such as velocity measurement or object trajectory prediction.

A common failure of the vision algorithm lies in the intolerance of objects bordering the camera's field of view. VISYS is occasionally subject to profound irrecoverable system failures when presented with images bordering the view area. This may be a weakness in Warkentin's original implementation of the Runlengthcoding algorithm.

The circle method used to determine orientation is highly sensitive to perimeter point resolution. The perimeter, essentially the endpoints of the runlengthcode algorithm, becomes sparse when a linear object edge is aligned with the scan lines of the camera. The result is that large gaps in the perimeter, intersected twice by the circular boundary, are represented only once as an intersection point. Though this is a rare occurrence, the result will often produce incorrect object orientation angles. Employing a higher order interpolation might be one solution, but this would lengthen the processing time in an already prolonged vision cycle. Future versions of VISYS should employ more robust orientation procedures than this prototype circle method, preferably one not relying on raw unsmoothed perimeter data. Emphasis should be placed on employing feature based orientation algorithms using corners and edges as references.

6.3 Recommendations for Future Research

6.3.1 Modifications to TSKMSTR

The following is a summary of the recommendations suggested in the above text.

Natural Language Modules

The natural language module should be modified by optimizing the syntactic grammar or developing a more descriptive semantic grammar. To reduce backtracking in the parser, a combined bottom–up/top–down parsing strategy should be adopted. The unification abilities of PROLOG should be applied to the semantic processing replacing the register method.

Furthermore, the Semantic Processor and the Parser should be unified into a single natural language module. The resultant increase in stack space should allow the module to generate *all* legal semantic interpretations for a command. These could be evaluated, in turn, by an intelligent command execution module.

Command Interpreter

Optimization of command template forms might lead to wider application of the interpreter system beyond the current creation of object relationships. Intelligent task planning, queuing and scheduling should be examined and incorporated into the interpreter's variable generalization mechanism. Further, the command interpreter should be modified to reduce the stack requirements of the PROLOG interpreter.

World Model

The teach environment should be equipped with a world model simulator. In order to create and debug accurate teach routines, some means of viewing and correcting teach procedures off line is required.

The kinematic inference engine can be further improved by implementing an interrupt driven world model, receiving continuous sensor data from the robot and vision system. Any mechanism which reduces the interval between world model updates increases the reliability of the world model. Currently vision and robot data refreshes the world model sporadically and only at the request of the system. Ideally this information should be continuously provided, analogous to a biological nervous system.

6.3.2 Other Approaches to Natural Language Interfaces and Work Cells

Future work on NLIs for robotics should investigate alternative processing algorithms. In particular the work of Selfridge et al. points to the effectiveness of the Conceptual Dependency analysis method. The success of TSKMSTR's largely syntactic grammar approach suggests further work should be done in the area of semantic or case grammars, similar to those in LIFER [19], for application to robotics.

Regardless of the natural language processing system, alternate world modelling methods should be investigated. In particular, solid object modelling methods such as CSG [28] and RAPT [24] type assembly models should be investigated. Further, the unification of the linguistic, kinematic, and visual world representations should be addressed. Though Selfridge et al. [16,17,18] have made some progress in this direction, the unification issues are far from resolved. Linguistic, kinematic, and visual world models are plainly related and interdependent in humans. Research should be directed towards resolving these different images of the same problem, namely communication between a sensor equipped work cell and a human operator.

6.4 A Final Comment on Programming Robots in English

What is the future for robots that understand English? To understand any human language, a robot must have some model of the human's world. Greater language comprehension in robots must, therefore, be mirrored by an equally well developed model of the human's environment. Natural language understanding is not the only field to benefit from a comprehensive world description, however. Task planning is equally reliant on a complete world model. One might suppose that as the world model becomes more sophisticated, the robot may become increasingly able to perform tasks autonomously, through task planning, and increasingly able to converse in some form of natural language.

It seems likely, therefore, that the abilities of a robot to perform tasks with little instruction may render the TSKMSTR form of NLI redundant since the robot would be capable of deducing the correct assembly instructions, as does a human, from a design schematic. Natural language instruction for assembly robots may, therefore, be limited to general plant scheduling.

Perhaps the greatest future of natural language processing combined with task planning might be seen in DWIM (Do What I Mean) and DIRN (Do It Right Now) class of commands, typical of remote manipulator systems. In this case the use of natural language would permit oral instructions while the robot's task planning mechanism frees the human operator from describing actions in tedious detail. A good example might be space-walking astronauts orally commanding an RMS manoeuvre, a task currently requiring the assistance of a human operator inside the shuttlecraft.

6.5 Conclusion

TSKMSTR, a prototype system, is capable of recognizing and executing user defined commands within a subset of Natural Language. Unlike it's predecessors, the system can be used to develop new 'generalized' commands through the use of a simple top down teaching method that generates and modifies meta-PROLOG predicates. Though the system's performance is nominally comparable to those outlined by Maas and Suppes, Selfridge et al., and Bock, linguistically TSKMSTR closely resembles an AUTOPASS system with an AL world model. Unlike AUTOPASS, however, TSKMSTR allows both on-line commands and off-line user defined subroutines to be expressed in a subset of English.

In general, the feasibility of a PC based natural language interface to a robotic work cell has been proven. As a basic system, TSKMSTR demonstrates the potential to produce a more useful and versatile assembly interface than currently available. Though the constraints of the modern PC are great, TSKMSTR establishes that they are clearly not insurmountable. Indeed, the fact that a PROLOG based system running:

- a relatively large prototype parser and semantic processor
- an AL–like world model
- an interactive teach interface
- a meta–PROLOG interpreter

functions effectively at all, should encourage industrial NLI developers to look at the PC as a serious interface alternative to Robotic Work Cells.

The system has been designed to investigate the minimum requirements of a Natural Language system. Plainly, syntax is a very useful tool to model much of the dialogue between man and machine. However as technology and modelling techniques improve, TSKMSTR's narrow AL–like world view may become too rigid for useful task description. A wider semantic groundwork should be laid for a full scale robotic NLI.

English is an imprecise method of describing the precise relationships between assembly components. However, it is also a powerful tool expressing, roughly, the method in which a task might be carried out. The challenge lies in designing software with the requisite intelligence to 'read between the lines' of assembly instructions. While this planning ability has not been treated here, the development of TSKMSTR has emphasized the need for research in not only natural language comprehension, but in semantic description and understanding of the world required for such planning skills. The development of TSKMSTR has pointed to a number of design alternatives in Work Cell/NLI design issues. Certainly there is considerable work yet to be done in this unusual and fascinating mixture of Robotics and Artificial Intelligence.

References

[1] E. Rich, *Artificial Intelligence*, McGraw Hill Book Company, pp. 295-334, 1983.

[2] S.E. Fahlman, *A Planning System for Robot Construction Tasks*, Artificial Intelligence 5, pp. 1-50, 1974.

[3] S. T. Rock *Intelligent Robot Programming: You Can't Get There From Here – A Viewpoint.* Robotica, Vol 6. , December, pp. 333-338, 1988.

[4] S. Bonner , K.G.Shin *A Comparative Study of Robot Languages* IEEE Computer, Vol 12., December, pp. 82-96, 1982.

[5] W. A. Gruver, B. I. Soroka, J. J. Craig, T. L. Turner, *Industrial Robot Programming Languages: A comparitive Evaluation.* IEEE Transactions on Systems, Man, and Cybernetics, August, pp. 565-570, 1984.

[6] T. Lozano-Perez and R.A. Brooks, *Task Level Manipulator Programming*, Handbook of Industrial Robotics, S.Y. Nof (edit.), John Wiley and Sons, 1985.

[7] T. Lozano-Perez, *Robot Programming*, Proceedings of the IEEE, Vol. 71, No. 7, pp. 821-841, 1983.

[8] L. I. Lieberman, M. A. Wesley, *AUTOPASS: An Automatic Programming System For Computer Controlled Assembly.* IBM Journal of Research and Development, July, pp. 321-333, 1977.

[9] M. S. Mujtaba, R. Goldman, T. Binford, *The AL Programming Language*, IEEE Tutorial on Robotics, 1986

[10] R. Goldman *Design of An Interactive Manipulator Programming Environment.*, UMI Research Press, 1985.

[11] K. Takase, R.P. Paul, E. Berg., *A Structured Approach to Robot Programming and Teaching.*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 11, April, pp. 274-289, 1981.

[12] V. Hayward, R.P. Paul, *Robot Manipulator Control under Unix RCCL: A Robot Control "C" Library*, The International Journal of Robotics Research, Vol. 5, No. 4, pp. 94-111, 1986.

[13] C. Blume , W. Jakob, J. Favaro *PASRO and C for Robots*, Springer Verlag, 1987.

[14] C. Blume , W. Jakob, *Programming Languages for Industrial Robots*, Springer-Verlag, 1986.

[15] P. Bock, *Controlling Robots with an English Like High- Level Hierarchichal Command Language (HIROB)*, IEEE Int. Conf. on Robotics and Automation, pp. 404-412, 1984

[16] J. Engelberg, A. Levas, M. Selfridge, *A Robust Natural Language Interface to a Robot Assembly System*, IEEE Int. Conf. on Robotics and Automation, pp. 400-403, 1984

[17] M. Selfridge, W. Vannoy, *A Natural Language Interface to Robot Assembly System*, IEEE Journal of Robotics and Automation, Vol. 2, No. 6., pp. 167-171, 1986.

[18] A. Levas, M. Selfridge, *A User-Friendly High-Level Teaching System*, IEEE Int. Conf. on Robotics and Automation, pp. 413-416, 1984.

[19] G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, J. Slocum, *Developing a Natural Language Interface to Complex Data*, ACM Transactions on Database Systems, Vol. 3, pp. 105-147, 1978.

[20] T. Winograd, *Understanding Natural Language*, Academic Press, 1972.

[21] R.E. Maas, P. Suppes, *Natural-language Interface for an Instructable Robot.*, Int. J. of Man-Machine Studies, Vol. 22, pp. 215-240, 1985.

[22] R. Paul, *Robot Manipulators: Mathematics, Programming, and Control.*, The MIT Press, 1981.

[23] A.P. Ambler, H.G. Barrow, C.M. Brown, R.M. Burstall, *A Versatile System For Computer Controlled Assembly.*, Artificial Intelligence, Vol. 6, pp. 129-156, 1975.

[24] A.P. Ambler, R.J. Popplestone, *Inferring the Positions of Bodies from Specified Spatial Relationships.*, Artificial Intelligence, Vol. 6, pp. 157-174, 1975.

[25] R.J. Popplestone, A.P. Ambler, I.M. Bellos, *An Interpreter for a Language for Describing Assemblies*, Artificial Intelligence, Vol. 14, pp. 79-107, 1980.

[26] A.P. Ambler, R.J. Popplestone, K.G. Kempf, *An Experiment in the Offline Programming of Robots.*, 12th Int. Symp. On Industrial Robotics, pp. 491-504, 1982.

[27] F. Thomas, C. Torras, *A Group Theoretic Approach to the Computation of Symbolic Parts Relations* IEEE Journal of Robotics and Automation, Vol. 4, No. 6, pp. 622-634, 1988.

[28] Aristides A. G. Requicha, *Representations for Rigid Solids: Theory, Methods, and Systems*, Computing Survey, Vol. 12, No. 3, pp. 437-464, 1980

[29] P. Freedman, C. Michaud, G. Carayannis, A. Malowany, *A Data Base Design for the Runtime Environment of a Robotic Workcell*, TR-CIM-87-21, Computer Vision and Robotics Laboratory, McGill Research Centre for Intelligent Machines, McGill University, January, 1988

[30] A.M. Segre, Gerald Dejong, *Explanation-Based Manipulator Learning: Acquisition of Planning Ability through Observation.*, IEEE Int. Conf. on Robotics and Automation, pp. 555-560, 1985.

[31] T. Lozano-Perez, M. Brady (Edit), *Task Planning*, Robot Motion Planning and Control, The MIT Press,1982.

[32] M.P. Groover, M. Weiss, R. Nagel, N.G. Odrey, *Industrial Robotics: Technology, Programming, and Applications*, McGraw-Hill Book Company, 1986.

[33] Dan Hays, *Communicating With Robots: Problems in Trans-Systems Linguistics.*, Papers On Computational and Cognitive Science, Edwin Battistella,(edit.), pp. 39-59,1984

[34] I. Lee, S.M. Goldwasser, *A Distributed Testbed For Active Sensory Processing.* IEEE Int. Conf. on Robotics and Automation, pp. 925-930, 1985.

[35] M. Gini, R. Doshi, M. Gluch, R. Smith, I. Zauldernan, *The Role of Knowledge in the Architecture of a Robust Robot Control*, IEEE Int. Conf. on Robotics and Automation, pp. 561-567, 1985.

[36] G.N. Saridis, *Robotic Control to Help the Disabled*, Recent Advances in Robotics, edit. G. Beni, S. Hackwood. John Wiley and Sons publishers, pp. 35-67, 1985.

[37] *Turbo C Manual Version 1.5*, Borland International Inc., 1987.

[38] *Turbo Pascal Manual Version 4.0*, Borland International Inc., 1987.

[39] I. Bratko, *Prolog Programming For Artificial Intelligence*, International Computer Science Series, Addison Wesley Publishing Co., 1986.

[40] *Turbo Prolog Manual Version 2.0*, Borland International Inc., 1988.

[41] *Turbo Prolog Toolbox*, Borland International Inc., 1987.

[42] Robotic Systems International, *Excalibur Users Manual*, Robotic Systems International, 1986.

[43] *Operators Manual for using the Micro D-cam with the IBM-PC*, Circuit Cellar and Micromint Inc. 1983.

[44] P. Tanisch, B. V. McNally, and A. Robin, *Linear Time Algorithm For Finding A Pictures Connected Components*, Image and Vision Computing, Vol. 2., No. 4, pp. 191-197,1984.

[45] A. Pugh, *Processing of Binary Images* Robotic Technology, pp. 63-87, 1983.

[46] Ming-Kuei Hu, *Visual Pattern Recognition By Moment Invariants*, IRE Transactions on Information Theory, vol IT-8, February, pp. 179-187, 1962.

[47] G. J. Agin, *Vision Systems* (Chapter 14), Handbook of Industrial Robotics. S.Y. Nof (edit.), John Wiley and Sons, pp. 231-261, 1985.

[48] T. Kasvand, *Vision as Applied to Robots*, ERB-932 NRCC NO. 21004, National Research Council Canada, Division of Electrical Engineering, January, 1983.

[49] J. Warkentin, *DCAMLIB Vision System Source Code*, University of Alberta, 1986.

[50] N. Chomsky, *Three Models for the Description of Language* , IRE Transactions, PGIT, 2, pp. 113-124, 1956.

[51] S. A. Mehdi, *Arabic Language Parser*, International Journal of Man-Machine Studies, Vol. 25, No. 5, pp. 593-611, 1986.

[52] D. C. Bennett, *Spatial and Temporal Uses of English Prepositions An Essay in Stratificational Semantics.*, pp. 1-92, Longman, 1975

[53] F. Oppacher, *Design Issues In Natural Language Processing Systems.*, Applied Systems and Cybernetics Proc. of the Int. Con. on Appl. Systems Research and Cybernetics, Vol. 5, pp. 2291-2297, 1981.

[54] F. C. N. Pereira, D. H. D. Warren, *Definite Clause Grammars for Language Analysis* Artificial Intelligence 13, pp. 231-278, 1980.

[55] B. Clinger, *Definite Clause Grammars in Turbo Prolog*, Turbo Technix The Borland Language Journal, Vol. 1, No. 5, pp. 80-88, 1988

[56] J. Allen, *Natural Language Understanding*, Benjamin Cummings Publishing Co., 1988

[57] R. C. Schank (edit), C. K. Riesbeck (edit), *Inside Computer Understanding: Five Programs Plus Miniatures*, Lawrence Erlbaum and Associates, pp. 1-40 and 318-353, 1981

[58] P. Roberts, *Understanding Grammar*, Harper and Brothers, Publishers, New York, 1954.

[59] E. Charniak, *The Case-Slot Identity Theory*, Cognitive Science 5, 3, pp. 285-292, 1981.

Appendix A

Robot Kinematics and Modelling

Most industrial robots are composed of a number of *links* in series. Each link is connected to adjoining links through *joints*. Though the links are often of any dimension and shape, joints tend to be one of two kinds: Prismatic (permitting only linear motion) or Revolute (permitting rotation about some axis). The variety, number, and placement of these joints affects the manipulator's ability to position and orient the final link or end effector within the total accessible volume, the work envelope.

To reach any point in a three dimensional work envelope, requires three translational degrees of freedom or a combination of at least three Revolute or Prismatic joints in a manipulator. To generate any orientation in three dimensional space requires three rotational degrees of freedom or a combination of at least three rotational joints. An efficient manipulator capable of reaching any orientation and position within a work envelope must have at least six joints or six degrees of freedom.

For the manipulator to be consistently placed within the work volume, a function must be developed relating the robot's joint angles to a position in space. This transformation from *joint space* to *cartesian space* is called the *forward* solution.

Though the forward solution for manipulators of two or three links can be derived through geometry, geometric methods become tedious and error prone as manipulators grow to six degrees of freedom. A swift and compact solution to this problem is presented by Paul [22] and is briefly summarized below.

A.1 Position

In it's simplest form a position in three-space can be represented as a vector relative to a known point. For example, a point, $a\mathbf{i}+b\mathbf{j}+c\mathbf{k}$, in a cartesian coordinate system, $\mathcal{XYZ}$, with can be described with a column vector :

$$\vec{p} = \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}$$

This column vector can be incorporated into a transformation matrix, P, such that a the product of P and a translation matrix, $\mathbf{Trans}(d,e,f)$, generates a new matrix P' describing another location in $\mathcal{XYZ}$. Symbolically ...

$$P \ \mathbf{Trans}(d,e,f) \ = \ P'$$

in matrix form ...

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & e \\ 0 & 0 & 1 & f \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & a+d \\ 0 & 1 & 0 & b+e \\ 0 & 0 & 1 & c+f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A position matrix postmultiplied by a $\mathbf{Trans}(x,y,z)$ expression represents a translation relative to that position by an amount x,y,z, in effect creating a new coordinate system at that point. Consecutive translations through 3 space are expressed by a chain of postmultiplied **Trans**() expressions.

A.2 Orientation

If the three axes of a local coordinate system are described by three mutually perpendicular unit vectors defined relative to a global coordinate system, the orientation of the local coordinate system can be specified by an orientation matrix, $\mathcal{R}$. For example, if the local axes $\mathcal{UVW}$ are described by three column vectors, $\vec{n}$, $\vec{o}$, and $\vec{a}$ (where $\vec{n} \times \vec{o} = \vec{a}$), relative to a global system $\mathcal{XYZ}$, a 3 × 3 matrix can be generated:

$$\mathcal{R} = \begin{bmatrix} n_x & o_x & a_x \\ n_y & o_y & a_y \\ n_z & o_z & a_z \end{bmatrix}$$

where the subscripts x, y, and z denote the direction cosines of each of the unit vectors $\vec{n}$, $\vec{o}$, and $\vec{a}$ in the global $\mathcal{XYZ}$ directions.

If this matrix is incorporated into a 4 × 4 transformation matrix, a Rotation Transformation is developed, describing rotation about some vector, $\vec{k}$, by an amount θ or:

$$\mathbf{Rot}(k,\theta) = \begin{bmatrix} n_x & o_x & a_x & 0 \\ n_y & o_y & a_y & 0 \\ n_z & o_z & a_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using this notation the three principle rotations (one about each axis) can be expressed as: $\mathbf{Rot}(x,\theta)$, $\mathbf{Rot}(y,\theta)$, and $\mathbf{Rot}(z,\theta)$ or :

$$\mathbf{Rot}(x,\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{Rot}(y,\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{Rot}(z,\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The product of a 4 × 4 matrix describing a coordinate system $\mathcal{UVW}$, and a postmultiplied rotation transformation, $\mathbf{Rot}(\vec{\imath},\theta)$ is equivalent to creating a new coordinate system rotated an amount θ about the vector $\vec{\imath}$ relative to $\mathcal{UVW}$. It has been shown in Paul [22], that the product of a series of rotations about a set of axes can be expressed as a single rotation about a unique vector. The resulting general expression is called $\mathbf{Rot}(\vec{k},\theta)$.

Since at least three successive rotations are required to model the orientation of a coordinate frame in three dimensional space, orientation can be expressed as the product of three $\mathbf{Rot}(\vec{\imath},\theta)$ terms. Two popular conventions modelling orientation are the Euler transform and the Roll Pitch Yaw (RPY) transform.

The Euler Transform is defined as the result of:

- a rotation about the z axis an amount ϕ followed by ...
- a rotation about the new y axis an amount θ followed by ...
- a rotation about the new z axis an amount ψ.

or

$$\mathbf{Euler}(\phi,\theta,\psi) = \mathbf{Rot}(z,\phi)\ \mathbf{Rot}(y,\theta)\ \mathbf{Rot}(z,\psi)$$

The RPY Transform is defined as the result of:

- a *roll* about the z axis an amount ϕ followed by ...
- a *pitch* about the new y axis an amount θ followed by ...
- a *yaw* about the new x axis an amount ψ.

or

$$\mathbf{RPY}(\phi,\theta,\psi) = \mathbf{Rot}(z,\phi)\ \mathbf{Rot}(y,\theta)\ \mathbf{Rot}(x,\psi)$$

A.3 The Homogeneous Transform

Combining the Translation transformation and the Rotation transformation produces the homogeneous transform,T.

$$\mathbf{T} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This transformation creates a new coordinate system at point $\vec{p}$, and with an orientation described by the vectors $\vec{n}$, $\vec{o}$, and $\vec{a}$. If a new coordinate system, C, is generated by the product of A and T or symbolically...

$$\mathbf{A\ T = C}$$

then A can be expressed as the product of the new system C and the inverse of transform T or ...

$$\mathbf{A = C\ T^{-1}}$$

The inverse of T being defined as the following...

$$\mathbf{T}^{-1} = \begin{bmatrix} n_x & n_y & n_z & -\vec{p}\cdot\vec{n} \\ o_x & o_y & o_z & -\vec{p}\cdot\vec{o} \\ a_x & a_y & a_z & -\vec{p}\cdot\vec{a} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Four alternative representations of position and orientation in three-D space are possible. See Table A.1.

A.4 Homogeneous Transforms and Robot Kinematics

If the base of a manipulator is placed at the origin of a global coordinate system, the position of the end effector can be expressed through an homogeneous transformation relative to the base coordinates. However, the position of the each joint is dependent on the position of the preceding joints of the manipulator. Therefore, the location of the end effector must be expressed in terms of the preceding joints' positions. The net result is a chain of kinematic transforms from the manipulator base to the end effector. The expression used to describe the joint-to-joint transformation is a narrow definition of the homogeneous transform widely referred to as the **A** matrix [22].

The **A** matrix is a particular series of rotations and translations that relate the location of an n^{th} link to the location of link $n-1$ within a multijointed manipulator. If a coordinate system is placed on each joint such that the z axis is aligned with a unique axis of motion (i.e. along the direction of translation for the prismatic joint or through a revolute joint's rotational axis), the **A** matrix relating the coordinate systems of link n with link $n-1$ is described as...

- a rotation θ_n about axis z_{n-1} followed by ...
- a translation d_n along axis z_{n-1} followed by ...
- a translation a_n along new axis x_n followed by ...
- a rotation α_n about new axis x_n.

Using traditional transform notation:

$$\mathbf{A}_n = \mathbf{Rot}(z, \theta_n)\ \mathbf{Trans}(z, d_n)\ \mathbf{Trans}(x, a_n)\ \mathbf{Rot}(x, \alpha_n)$$

which becomes...

$$\mathbf{A}_n = \begin{bmatrix} \cos\theta_n & -\sin\theta_n \cos\alpha_n & -\sin\theta_n \sin\alpha_n & a_n \cos\theta_n \\ \cos\theta_n & \cos\theta_n \cos\alpha_n & -\cos\theta_n \sin\alpha_n & a_n \sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For a revolute joint, the characteristics of α, d, and a are fixed while for a prismatic manipulator θ and a are fixed. Once the fixed parameters have been tabulated the **A** matrix for each link can be evaluated.

Using an **A** matrix to place the location of the joint 1 relative to joint 0, the base joint, results in the simple equation :

$$^0\mathbf{T}_1 = \mathbf{A}_1$$

- where the subscript on **A** denotes the joint number and
- where the superscript on **T** denotes the start of the

kinematic chain, joint 0, and the subscript denotes the last joint or frame on the right hand side, joint 1.

Subsequent **A** matrices are then postmultiplied to the equation, successively expressing each joint in base coordinates.

$$\begin{aligned} {}^0T_2 &= \mathbf{A}_1\mathbf{A}_2 \\ {}^0T_3 &= \mathbf{A}_1\mathbf{A}_2\mathbf{A}_3 \\ {}^0T_4 &= \mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4 \\ &\vdots \\ T_n &= \mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\ldots\mathbf{A}_{n-1}\mathbf{A}_n \end{aligned}$$

Until, finally, the right hand side is postmultiplied by the final joint, joint n, producing the $\mathbf{T}_n$ matrix, where n is the number of joints in the manipulator. Note that the zero superscript, indicating a transformation relative to base coordinates, is sufficiently common that it is usually dropped.

A T_n matrix is referred to as the *forward* kinematic solution, since it expresses the cartesian end effector position and orientation in terms of the manipulator's joint space.

An important byproduct of forward kinematic equations is the *inverse* kinematic solution. The inverse solution determines the manipulator joint angles that provide a desired end effector position and orientation. Although this solution can be generated geometrically, the solution procedure becomes complex for more than 3 or 4 links. One alternate method, outlined by Paul [22], involves the successive decomposition of the right hand side of the forward solution from the base of the manipulator to the end effector. This procedure, illustrated by the following equations, successively isolates single unknown joint angles to the left hand side.

$$\begin{aligned}
\mathbf{A}_1^{-1}T_n &= \mathbf{A}_2\mathbf{A}_3\ldots\mathbf{A}_{n-1}\mathbf{A}_n \\
\mathbf{A}_2^{-1}\mathbf{A}_1^{-1}T_n &= \mathbf{A}_3\ldots\mathbf{A}_{n-1}\mathbf{A}_n \\
\mathbf{A}_3^{-1}\mathbf{A}_2^{-1}\mathbf{A}_1^{-1}T_n &= \ldots\mathbf{A}_{n-1}\mathbf{A}_n \\
&\vdots \\
\mathbf{A}_{n-1}^{-1}\ldots\mathbf{A}_3^{-1}\mathbf{A}_2^{-1}\mathbf{A}_1^{-1}T_n &= \mathbf{A}_n
\end{aligned}$$

The resulting solution of these equations does not produce an elegant matrix, like the T_n forward solution. The inverse solution may produce redundant joint space positions and often the solution strategy for one joint may be dependent on the outcome of another.

In summary the Forward solution can be used to determine the cartesian location and orientation of the end effector based on the robot's joint angles, while the Inverse solution provides a means to determine the joint angles necessary to place the gripper in a predetermined position and orientation in the work envelope.

Table A.1: Alternate Representations of Position and Orientation

Position	Orientation	Representation	
vector	**Euler**	$\vec{p}$	$\langle \phi, \theta, \psi \rangle$
vector	**RPY**	$\vec{p}$	$\langle \phi, \theta, \psi \rangle$
vector	Orientation Matrix	$\vec{p}$	$[\vec{n}\ \vec{o}\ \vec{a}]$
Homogeneous Transform		$[\vec{n}\ \vec{o}\ \vec{a}\ \vec{p}]$	

Appendix B

Binary Vision Processing Algorithms

In this appendix, the binary vision processing techniques employed by VISYS will be summarized.

B.1 The Moments of Area

A general expression for the moment of area, M_{pq}, is described by both Ming-Kuei Hu [46] and the Handbook of Robotics [47] as:

$$M_{pq} = \sum_{i=1}^{N} x^p y^q$$

where N is the total number of pixels and where the sum of p and q denotes the order of the moment of area.Note that x and y are the location of the *on* pixels in the image. The area of a blob is equivalent to the zeroth moment of area, M_{00}, or:

$$A = M_{00} = \sum_{i=1}^{N} 1 = N$$

Only two first moments of area are possible namely :

$$M_x = M_{01} = \sum_{i=1}^{N} x$$

and

$$M_y = M_{10} = \sum_{i=1}^{N} y.$$

The second moments of area, the moments of inertia, are expressed by the following three equations:

$$I_x \;=\; M_{20} = \sum_{i=1}^{N} x^2$$

$$I_y = M_{02} = \sum_{i=1}^{N} y^2$$

$$I_{xy} = M_{11} = \sum_{i=1}^{N} xy$$

The centroids of the blobs can be determined by the relations:

$$\overline{x} = \frac{M_{10}}{M_{00}} \quad \text{and} \quad \overline{y} = \frac{M_{01}}{M_{00}}$$

The moments of inertia can then be recomputed about the centroid through the parallel axis theorem:

$$\overline{I}_x = I_x - A\overline{x}^2 \quad \overline{I}_y = I_y - A\overline{y}^2 \quad \overline{I}_{xy} = I_{xy} - A\overline{xy}$$

Though the general expression for the centroidal moments, μ'_{pq} is:

$$\mu'_{pq} = \sum_{i=1}^{N} (x - \overline{x})^p (y - \overline{y})^q$$

these values are also obtainable simply through the substitution of $\overline{x}$, $\overline{y}$, A, I_x and I_y i.e.

$$\overline{I}_x = \mu'_{20} = M_{20} - \frac{M_{10}^2}{M_{00}}$$

$$\overline{I}_y = \mu'_{02} = M_{02} - \frac{M_{01}^2}{M_{00}}$$

$$\overline{I}_{xy} = \mu'_{11} = M_{11} - \frac{M_{10}M_{01}}{M_{00}}$$

Similar expressions can be generated for the higher moments of area such as μ'_{21}, μ'_{12}, μ'_{30} and μ'_{03}.

Though higher moments of inertia are often generated for moment Invariants, VISYS uses only the first three moments of inertia.

B.2 Moment Invariants

B.2.1 Scale

To normalize these central moments to image size the following transformation from [46] is performed:

$$\mu_{pq} = \left[\frac{\mu_{00}}{\mu'_{00}}\right]^{\frac{p+q}{2}+1} \mu'_{pq} \tag{B.1}$$

Typically a standard area, μ_{00}, of unity is used, producing the final equation:

$$\mu_{pq} = \left[\frac{1}{\mu'_{00}}\right]^{\frac{p+q}{2}+1} \mu'_{pq} \tag{B.2}$$

B.2.2 Rotation

Though Ming Kuei Hu [46] develops six rotation invariants, in practice only the first three are required for identification. All employ the scale invariant quantities derived through the transformation above. The first three invariant expressions are

$$I_1 = \mu_{20} + \mu_{02} \tag{B.3}$$

$$I_2 = (\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2 \tag{B.4}$$

$$I_3 = (\mu_{30} - 3\mu_{12})^2 + (3\mu_{21} - \mu_{03})^2 \tag{B.5}$$

B.3 VISYS Shape Descriptors

VISYS employs five shape descriptors:

- The Area
- The First Moment Invariant, I_1

- The Number of Holes
- The Perimeter Length
- The Density
- The Compactness, S_1

B.3.1 The Area and First Moment Invariant, I_1.

After the image has been acquired by the optic RAM cameras and enhanced through, growing and shrinking techniques, the image is converted into a runlengthcode array structure. This structure is then evaluated for connectivity using the algorithm published by Thanisch et al [44].

The resulting structure is a linked list of runlengthcode records. Each record contains the location of the start and end pixels for a scan line and a pointer to a neighbouring blob record. For the following discussion the runlengthcode array may be modelled as the following data structure:

```
RunRecord = array[1..SomeInteger] of record
                                        lhs,
                                        rhs,
                                        object,
                                        scan,
                                        rnext : Integer;
                                    end;
```

The terms `lhs` and `rhs` represent the left and right hand sides of a consecutive row of pixels, while the `object` is an object label, `scan` is the scan line number, and `rnext` is the array index of an adjoining runrecord element. The area and moment invariants are computed as described below (condensed from [47] and [49]), modified by the scale factor S_x and S_y relating image and real world dimensions. The runlengthcode elements become:

$$m_j = \texttt{Run[j].lhs} \quad \text{and} \quad n_j = \texttt{Run[j].rhs} \quad \text{and} \quad y_j = \texttt{Run[j].scan}$$

The Area becomes:

$$M_{00} = S_x S_y \sum_{j=1}^{k} n_j - m_j$$

The first moments of area become:

$$\begin{aligned} M_{10} &= S_x^2 S_y \sum_{j=1}^{k} \frac{1}{2}(n_j - m_j)(m_j + n_j - 1) \\ M_{01} &= S_x S_y^2 \sum_{j=1}^{k} y_j (n_j - m_j) \end{aligned}$$

The second moments of area become:

$$\begin{aligned} M_{20} &= S_x^3 S_y \sum_{j=1}^{k} \frac{1}{12} \left[3(n_j - m_j)(n_j + m_j - 1)^2 + (n_j - m_j)^3 - (n_j - m_j)\right] \\ M_{02} &= S_x S_y^3 \sum_{j=1}^{k} y_j^2 (n_j - m_j) \\ M_{11} &= S_x^2 S_y^2 \sum_{j=1}^{k} \frac{1}{2}(n_j - m_j)(m_j + n_j - 1) y_j \end{aligned}$$

similar expressions may be derived for M_{30}, M_{03}, M_{21}, and M_{12}. After normalization for scale using equation B.2 the first moment invariant, using equation B.3, may be determined. These moments may also be used to develop an orientation criteria through the determination of the principal axes on the Mohr's Circle. Caution should be used, however, to employ this orientation method only on objects of even rotational symmetries. An object's principal axis does not provide a unique object orientation, since a rotation of either θ or $\theta + \pi$ will rotate the inertia axes to their principal values.

B.3.2 Density

The Ideal Ellipse is often used in binary vision processing to provide shape descriptors to those described above. The properties of an ellipse with identical moments to the object are determined about the object's centroid. These properties include the Major and Minor axis dimensions, the rotation of the ellipse. VISYS, however, uses only the ellipse density, the ratio of blob to ellipse area, as an additional shape descriptor.

$$\begin{aligned}
A &= \frac{4}{\pi}(M_{20} - \frac{M_{10}^2}{M_{00}}) \\
B &= \frac{4}{\pi}(M_{02} - \frac{M_{01}^2}{M_{00}}) \\
C &= \frac{4}{\pi}(M_{11} - \frac{M_{10}M_{01}}{M_{00}}) \\
E &= \sqrt{(A-B)^2 + \frac{4}{\pi}C^2} \\
F &= \sqrt{AB - C^2}
\end{aligned}$$

Where the expression for density becomes

$$\text{Density} = \frac{4}{\pi}\frac{M_{00}}{2F}$$

B.3.3 The Perimeter Length, Number of Holes and Compactness

During image processing, VISYS constructs a perimeter list that contains the location of all the pixels describing the edges and holes on an object. The format of the perimeter list is an array containing the following information:

Index 0 : The number of edges traced.

Index 2 : The Index of edge 1.

Index 4 : The Index of edge 2.

... : etcetera

Index 18 : The number of horizontal pixels in perimeter

Index 20 : The number of vertical pixels in perimeter

Index 22 : The number of diagonal pixels in perimeter

Index 24 :The start of perimeterlist outer boundary.

Number of Holes

The number of edges minus 1 (the outer boundary) is the number of holes in the object. The area and centroids of a hole are developed through a numerical integration technique. Refer to figure B.1. It is important to remember that the n perimeter points are recorded counterclockwise in camera coordinates.

Area The area, A, of a hole is defined as

$$A = \int y \, dx = \sum_{i=1}^{n} \frac{(y_{i+1} + y_i)}{2}(x_i - x_{i+1})$$

Since $(x_i - x_{i+1})$ is negative as the perimeter advances in the positive x direction, this has the effect of *removing* area beneath the (x_i, x_{i+1}) segment. However, $(x_i - x_{i+1})$ is positive in the negative x direction, therefore adding area beneath the segment (x_i, x_{i+1}). Note that at $i = n$, $i + 1$ is set to 1.

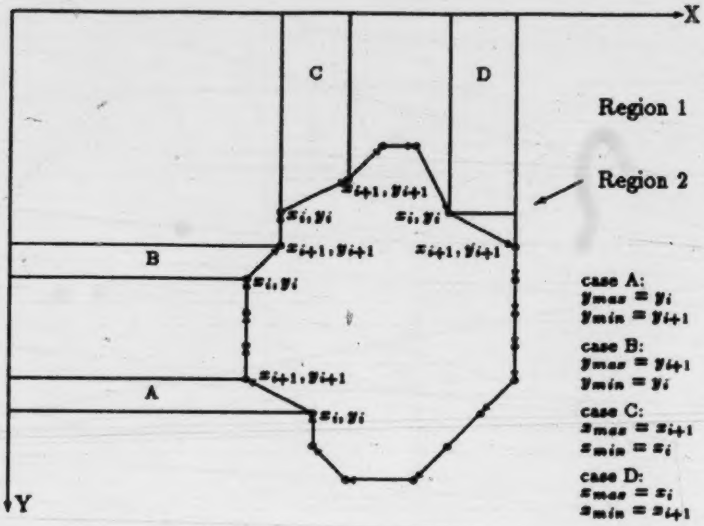

X
Y
C
D
Region 1
Region 2
B
A
x_{i+1}, y_{i+1}
x_i, y_i
x_i, y_i
x_{i+1}, y_{i+1}
x_{i+1}, y_{i+1}
x_i, y_i
x_{i+1}, y_{i+1}
x_i, y_i
case A:
$y_{max} = y_i$
$y_{min} = y_{i+1}$
case B:
$y_{max} = y_{i+1}$
$y_{min} = y_i$
case C:
$x_{max} = x_{i+1}$
$x_{min} = x_i$
case D:
$x_{max} = x_i$
$x_{min} = x_{i+1}$

Figure B.1: A hole perimeter in camera coordinates. Arrows indicate direction of perimeter list.

First Moment of Area A similar technique is adopted for the first moment of area.

For M_x:

$$\begin{aligned} M_x &= \int y \, dA \\ &= \sum y_1 A_1 + y_2 A_2 \\ &= \sum_{i=1}^{n} \left[\left(\frac{(y_{max} - y_{min})^2}{6} + \frac{y_{min}}{2}(y_{max} - y_{min}) + \frac{y_{min}^2}{2} \right) (x_i - x_{i+1}) \right] \end{aligned}$$

and for M_y:

$$\begin{aligned} M_y &= \int x \, dA \\ &= \sum x_1 A_1 + x_2 A_2 \\ &= \sum_{i=1}^{n} \left[\left(\frac{(x_{max} - x_{min})^2}{6} + \frac{x_{min}}{2}(x_{max} - x_{min}) + \frac{x_{min}^2}{2} \right) (y_{i+1} - y_i) \right] \end{aligned}$$

where max and min subscript implies the maximum x or y values in a pair of x, y coordinates.

Perimeter Length

The perimeter's actual length is then described by:

$$\text{Perimeter} = S_x N_x + S_y N_y + \sqrt{S_x^2 + S_y^2}\, N_d$$

where S_x and S_y are scale factors relating pixel dimensions to view area dimensions in the x and y directions respectively while N_x, N_y, and N_d are the number of pixels in the x, y, and diagonal directions.

Compactness

The compactness of an object is the ratio of object area to perimeter length is described by Groover et al [32] as:

$$S_1 = \frac{\text{Perimeter}^2}{\text{Area}}$$

and has been similarly defined by Warkentin [49] as:

$$S_1 = \frac{\text{Perimeter}^2}{4\pi\,\text{Area}}$$

These shape descriptors are stored in a Shape File during an editing session in the Shape Editor and consulted during the VISYS identification cycle.

Appendix C

An Explained Dialogue Session

The following is a complete dialogue with examples of the intermediate language processing. The contents of the Command, System Messages, and Teach interface windows will be provided and explained.

User: `Grasp the head of the bolt.` [1]

System: `Syntax: ok`

`Unknown Task: grasp Do you wish to enter Teach mode?(y/n)` [2]

`In general, would you say that you can` *grasp* `anything?(y/n)` [3]

`Then would you say that ...`

`you can` *grasp* `ANYTHING in this manner not just a` *head*`?(y/n)` [4]

`...of any object not just a` *bolt*`?(y/n)` [5]

---

[1] Once entered by the user in the Command window, the command is parsed. If successful the message `Syntax: ok` appears in the message window. The semantic processor reduces this command to the template:

`com(v(grasp,□),nil,f(head,[loc(of,bolt,□)]),nil)`

This template is then compared against a library of meta-PROLOG predicates.

[2] In this case, no match for this *grasp* template is found. Upon answering 'yes', a teach window is invoked and the query engine started.

[3] A 'no' answer would indicate that this is a unique case and is not to be generalized, shutting down the query engine. The user's answer, 'yes', indicates that to some degree the task is generalizable, invoking a stream of questions related to the command.

[4] A negative answer would 'lock' the direct object field into a constant value. The user's positive answer, however, generalizes the direct object noun and will convert all future occurances of this noun in the teach session into variable arguments.

[5] Similarly, the user's affirmative answer to this question generalizes all cases of *bolt*, in effect generalizing the adjective list. Since no more fields are present for generalization (`nil` indicates an empty field), the query session ends.

User: **Open the gripper.** [6]

System: **Syntax: ok**

User: **Move the gripper to the head of the bolt.**[7]

System: **Syntax: ok**

User: **Close the gripper.** [8]

System: **Syntax: ok**

User: **Affix the gripper to the head.** [9]

System: **Syntax: ok**

Meaning Unclear: Do you mean the *head* of the *bolt?*(y/n) [10]

User: **end** [11]

---

[6] The resulting command template is :

```
com(v(open,□),nil,f(gripper,□),nil)
```

Since this command exists in the library, only the "**Syntax: ok**" comment appears in the System window.

[7] The command string is legal and acknowledged, once again, by the "**Syntax: ok**" response. The resulting command template:

```
com(
    v(move,□),
    nil,
    f(gripper,□),
    tskpln(□,□,[plan(to,head,[loc(on,bolt,□)])])
    )
```

is successfully matched against the **move** library command.

[8] The syntax is acknowledged as legal and produces the following command template:

```
com(v(close,□),nil,f(gripper,□),nil)
```

and matched against the `close` library command.

[9] Since this command is incompletely specified (**head** is unmodified), no match will be made with a library object, thus spurring the question...

[10] A negative response produces a series of identical questions concerning other objects possessing the **head** noun, such as:

Meaning Unclear: Do you mean the *head* of the *nail*?(y/n)

An affirmative response resumes the semantic processing. Future unmodified occurances of the term *head* will result in the different question:

Meaning Unclear: Assuming you mean the *head* of the *bolt*?(y/n) thus avoiding a long list of incorrect trial questions by the system. The resulting command template:

```
com(
   v(affix,□),
   nil,
   f(gripper,□),
   tskpln(□,□,[plan(to,head,[loc(on,bolt,□)])])
   )
```

is then produced and matched against the **affix** library command.

[11] The **end** keyword completes the teach session and results in the assertion of a meta–PROLOG predicate into TSKMSTR memory. Note that this command definition will only be temporarily saved in the interpreter memory and will not be present if TSKMSTR is shut down and restarted. Command definitions may be permanently saved in files through the **save** keyword and retrieved in later sessions with the **consult** keyword. The saved true PROLOG predicate for the **grasp** command definition appears below:

```
grasp(□,nil,f(HEAD,BOLT),nil) :-
     open(□,nil,f("gripper",□),nil),
     move(□,nil,f("gripper",□),tskpln(□,□,[plan("to",HEAD,BOLT)])),
     close(□,nil,f("gripper",□),nil),
     affix(□,nil,f("gripper",□),tskpln(□,□,[plan("to",HEAD,BOLT)])).
```

In PROLOG capital letters indicate variable arguments. Note that they retain their original names, reminding the user of the origin of task and that the **gripper** is 'locked–in' as a constant. Note further that the conjunction relations have been preserved in the task plan fields.

TSKMSTR permits limited recursion through the repetition of the clause head in the body of the clause (or the body of some sub plan). For example a repetitive squeeze function with variable units might be modelled as:

```
System:  Execute?(y/n) [12]
         ⟨ open gripper 100 ⟩ OPEN 100
         * consulting Vision System ...
         ⟨ capture image ⟩ 1
         ⟨ camera 2 :  exposure 100 ms ⟩ 2 100
         ⟨ number of objects ⟩ 1
         1 NUT 1.296E+02 4.570E+02 1.023E+010K [13]
         * target NOT affixed [14]
```

```
squeeze([],nil,f("units",[NUM5]),nil) :-
        open([],f("gripper",[]),f("units",[NUM5]),nil),
        close([],f("gripper",[]),f("units",[NUM5]),nil),!,
        squeeze([],nil,f("units",[NUM5]),nil).
```

Where `NUM5` indicates that a variable number of units (originally 5 units in the teach session) may be used. It is important to note that the meta–cut operator, '!', is unable to prevent meta–PROLOG stack overflow and will only be effective in a true PROLOG environment.

[12] Answering 'yes' to this question executes the taught routine.

[13] The world model is searched and no *bolt* found. The vision system is subsequently invoked. Only one object, a nut, is in the field of view and inserted into the world model as:

```
arc(node(world,world),fixed,<sometransform>,fixed,node(W.nut,world)
arc(node(W.nut,world),unfixed,<ATtransform>,fixed,node(nut,nut))
```

By assuming the direction of the x-axis is the orientation angle and that the z-axis faces outward from the bottom of the nut's centroid an homogeneous transformation may be formed and placed in the `<sometransform>` field and labelled as `W.nut`.

[14] Since the `target`, in this case the `bolt`, was not found in the world model or in the field of view, the above message is displayed in the System Window reporting the failure of the user's command.

[15] This command, producing the following template:

```
com(v(grasp,[]),nil,f(nut,[loc(of,nut,[])]),nil)
```

matches the previously taught template for `grasp`. The term `f(nut,[loc(of,nut,[])])` is unified

User: `Grasp the nut.` [15]

System: `⟨ open gripper 100 ⟩ OPEN 100`

`⟨ move command ⟩ MOVETO 8752 522 60 -10 6453 -127654 0`

`⟨ close gripper 100 ⟩ CLOSE 100`

`* gripper affixed to nut` [16]

---

with the argument `f(HEAD,BOLT)` where all occurances of `HEAD=nut` and `BOLT=[loc(of,nut,☐)]`.

[16] The command is successfully executed and the gripper affixed to the nut. As the `move` command is processed, the symbolic task equation describing the manoeuver is placed in the Command window:

`T6 = W.nut · i · (i)^`

where `(i)^` indicates an inverted *at* transformation (an identity matrix), a product of the *to* conjunction in the `move` command. Note that TSKMSTR does not actually perform identity inversions or multiplications.

Appendix D

The BNF Grammar for the TSKMSTR Natural Language Interface.

The following is the BNF grammar used as input for the parser generator [41]. This grammar is largely based Bock's [15] context free grammar and has been adapted for use in the TSKMSTR environment. Note that some constructs presented here are not supported by either the semantic processor or the command interpreter but have been retained for future support and investigation.

```
COMMANDSTREAM     →   COMMAND CONN COMMANDSTREAM |
                      COMMAND period

COMMAND           →   MOD IMP-PRED MOD |
                      MOD IMP-PRED |
                      IMP-PRED MOD |
                      IMP-PRED

CONN              →   , coordinate-conjunction |
                      , |
                      coordinate-conjunction

MOD               →   MODGROUP CONN MOD |
                      MODGROUP MOD |
                      MODGROUP comma |
                      MODGROUP

IMP-PRED          →   COM-VERB-PHR I-OBJECT D-OBJECT |
                      COM-VERB-PHR D-OBJECT ADJ |
                      COM-VERB-PHR D-OBJECT |
                      COM-VERB-PHR

MODGROUP          →   PREP-PHRASE ADV-EXP |
                      PREP-PHRASE |
                      ADV-EXP

COM-VERB-PHR      →   COM-VERB-EXP CONN COM-VERB-PHR |
                      COM-VERB-EXP

I-OBJECT          →   NOUN-PHRASE CONN I-OBJECT |
                      NOUN-PHRASE

D-OBJECT          →   NOUN-PHRASE CONN D-OBJECT |
                      NOUN-PHRASE

ADV-EXP           →   subordinate-conjunction DEP-CLAUSE |
                      ADVERB

COM-VERB-EXP      →   VERB ADJ |
                      VERB
```

Table D.1: Backus Naur Form for the Natural Language Interface.

```
NOUN-PHRASE   →  article DES-PHRASE NOUN-GROUP |
                 article NOUN-GROUP |
                 DES-PHRASE NOUN-GROUP |
                 NOUN-GROUP

PREP-PHRASE   →  SOURCE PATH GOALS |
                 SOURCE GOALS |
                 SOURCE |
                 GOALS |
                 LOCATION

SOURCE        →  from NOUN-PHRASE

PATH          →  ONEPATH CONN PATH |
                 ONEPATH PATH |
                 ONEPATH

GOALS         →  ONEGOAL CONN GOALS |
                 ONEGOAL GOALS |
                 ONEGOAL

LOCATION      →  ONELOCATION CONN LOCATION |
                 ONELOCATION LOCATION |
                 ONELOCATION

ONEPATH       →  path-preposition NOUN-PHRASE

ONEGOAL       →  goal-preposition NOUN-PHRASE

ONELOCATION   →  locative-preposition NOUN-PHRASE

DES-PHRASE    →  DES-EXP CONN DES-PHRASE |
                 DES-EXP DES-PHRASE |
                 DES-EXP

NOUN-GROUP    →  NOUN LOCATION |
                 NOUN

DEP-CLAUSE    →  DEC-CLAUSE CONN DEP-CLAUSE |
                 DEC-CLAUSE
```

Table D.2: Backus Naur Form for the Natural Language Interface (cont.)

```
DES-EXP         →  ADJ |
                   ADVERB

DEC-CLAUSE      →  MOD SUBJECT PREDICATE |
                   SUBJECT PREDICATE

SUBJECT         →  NOUN-PHRASE CONN SUBJECT |
                   NOUN-PHRASE

PREDICATE       →  T-PRED MOD |
                   T-PRED |
                   INT-PRED MOD |
                   INT-PRED

T-PRED          →  T-VERB-PHR I-OBJECT D-OBJECT |
                   T-VERB-PHR D-OBJECT ADJ |
                   T-VERB-PHR D-OBJECT |
                   T-VERB-PHR

INT-PRED        →  INT-VERB-PHR NOUN-PHRASE |
                   INT-VERB-PHR DES-PHRASE

T-VERB-PHR      →  T-VERB-EXP CONN T-VERB-PHR |
                   T-VERB-EXP

INT-VERB-PHR    →  INT-VERB-EXP CONN INT-VERB-PHR |
                   INT-VERB-EXP

INT-VERB-EXP    →  AUX-EXP VERB |
                   VERB |

T-VERB-EXP      →  AUX-EXP COM-VERB-EXP |
                   COM-VERB-EXP
```

Table D.3: Backus Naur Form for the Natural Language Interface (cont.)

NOUN → noun | ambiguous-word |
coordinate-space VECTOR

VERB → verb | ambiguous-word

ADJ → adjective | ambiguous-word | real

ADVERB → adverb | ambiguous-word

AUX-EXP → auxiliary-verb auxiliary-verb |
auxiliary-verb

VECTOR → LPAR real, real, real, real, real, real, real RPAR |
LPAR real, real, real RPAR |
LPAR real, real RPAR |
LPAR real RPAR

LPAR → ⟨ | [| (|{

RPAR → ⟩ |] |) |}

Table D.4: Backus Naur Form for the Natural Language Interface (concluded).